

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！

Principles and Practice of Elasticsearch

Elasticsearch

技术解析与实战

朱林 编著

包含Elasticsearch 5最新功能，凝聚了作者多年开发经验

分布式大数据全文搜索与数据挖掘必备工具



机械工业出版社
China Machine Press

作者简介

朱林 资深开发人员，有16年开发经验，11年安全产品开发经验，对安全技术、日志分析有较深的研究。于2013年创立南京赛克蓝德网络科技有限公司，公司专注于安全产品的开发，目前主要开发的产品是赛克蓝德日志分析软件。

作者邮箱: zhulin@secisland.com

数据分析与决策技术丛书

Elasticsearch 技术解析与实战

朱林 编著

Elasticsearch 是目前全球最受欢迎的全文搜索引擎。初识 Elasticsearch 是在 2012 年的一个项目中。当时 Elasticsearch 还是 0.19.0 版本，但是功能已经比较强大，只是接口稍微有点复杂。到了 2015 年年初，公司开发了一款日志分析产品，它实时不间断地采集用户网络中各种不同系统的日志，然后从中分析系统的安全情况、系统情况、业务情况。最初所有的数据都有存储在 MySQL 中，随着日志的不断增长，MySQL 数据库增长速度超慢。后来在更换技术架构的时候想到了 Elasticsearch，这个时候 Elasticsearch 已经是 1.6.0 版本了。我们对此进行了简单的测试，在上亿条的数据搜索中很多都在一秒内完成，在上亿条的数据中进行统计分析大多也是在秒级完成，它展示了强大实力。我们赶紧就把 Elasticsearch 整合到了我们的产品中，取得了很好效果。到了 2016 年 3 月的时候，Elasticsearch 发布了 2.3.0 版本，各方面更加成熟，我们的产品又一次升级到这个新版本上。

Elasticsearch 产品的更新变化非常快，在我们开发研究的过程中基本上找不到新版本的中文资料。目前市场上介绍 Elasticsearch 的中文书籍都是在版本 1.0 左右，甚至更早，这些书有很多内容尤其是开发接口相关的部分已经过时了。在开发过程中经常遇到很多问题，有的时候甚至遇到文档都没有的问题。在遇到这样的问题的时候我们在 HTTP JSON 接口中进行了探索，我们发现了 Elasticsearch 的接口还是遵循 RESTful 风格，只是接口中多了很多参数。在开发过程中经常遇到一个问题，就是接口中缺少一个接口，三思而后行，开始更有利。比如添加接口的时候，我们经常会遇到一个问题，我们基本上总是研究怎么调用 Elasticsearch 的 API，但是研究怎么调用 Elasticsearch 的 API 还是有规律的，所以将 HTTP 接口转换成 Elasticsearch 的开发和调用也有很大的帮助。在研究 Elasticsearch 的过程中我们积累了大量的经验，我们把这些经验分享给更多的人。后来我把这个想法整理成书，供大家参考，也非常支持我们的团队，便有了这本书。



机械工业出版社
China Machine Press

图书在版编目 (CIP) 数据

Elasticsearch 技术解析与实战 / 朱林编著. —北京: 机械工业出版社, 2016.12
(数据分析与决策技术丛书)

ISBN 978-7-111-55327-4

I. E… II. 朱… III. 互联网络—情报检索 IV. G354.4

中国版本图书馆 CIP 数据核字 (2016) 第 274894 号

Elasticsearch 技术解析与实战

出版发行: 机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码: 100037)

责任编辑: 吴 怡

责任校对: 殷 虹

印 刷: 北京市荣盛彩色印刷有限公司

版 次: 2017 年 1 月第 1 版第 1 次印刷

开 本: 186mm×240mm 1/16

印 张: 27.25

书 号: ISBN 978-7-111-55327-4

定 价: 79.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88379426 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问: 北京大成律师事务所 韩光 / 邹晓东

Preface 前言

Elasticsearch 是目前全球最受欢迎的全文搜索引擎。初识 Elasticsearch 是在 2012 年的一个项目中，当时 Elasticsearch 还是 0.19.0 版本，但是功能已经比较强大，只是接口稍微有点复杂。到了 2015 年年初，公司开发了一款日志分析产品，它实时不间断地采集用户网络中各种不同系统的日志，然后从中分析系统的安全情况、系统情况、业务情况。最初所有的数据都存储在 MySQL 中，随着日志的不断增加，MySQL 搜索速度越来越慢。后来在更换技术架构选型的时候又想到了 Elasticsearch，这个时候 Elasticsearch 已经是 1.6.0 版本了。我们对此进行了简单的测试，在上亿条的数据搜索中很多都在一秒内完成，在上亿条的数据中进行统计分析大多也是在秒级完成，它展示了强大实力。我们顺势就把 Elasticsearch 整合到了现在的产品中，取得了很好效果。到了 2016 年 3 月的时候，Elasticsearch 发布了 2.3.0 版本，各方面更加成熟，我们的产品又再一次升级到这个新版本上。

Elasticsearch 产品的更新变化非常快，在我们开发研究的过程中基本上找不到新版本的中文资料，目前市场上介绍 Elasticsearch 的中文书籍都是在版本 1.0 左右，甚至更早，这些书的很多内容尤其是开发接口相关的部分都已经过时，没有办法在新版本中使用。所以我们开发的过程中基本上都是研究官方文档，有时候甚至研究它的源码才能解决问题。在接口选择的时候我们在 HTTP JSON 接口和 Java 接口中做了取舍，我们当时分析 HTTP JSON 接口最终还是要转换成 Java 接口，不如直接使用 Java 接口，一是效率可能更高，二是在部署实施的时候减少一个端口，三是对后续的升级更有利，比如后续增加权限认证等。但这些东西都没有资料，我们基本上都是研究系统源码来克服的。在后续研究过程中，我们发现 HTTP 接口转换到 Java 接口是有规律的，所以对 HTTP 接口的掌握对后续 Elasticsearch 的开发和扩展也有很大的帮助。在持续研究的过程中，我们积累了大量经验，并想把这些经验分享给更多需要的人。后来我把这个想法给出版社的吴怡编辑做了沟通，她非常支持我们的想法，便有了这本书。

本书首先介绍 Elasticsearch 的相关基础知识，然后由浅入深地介绍 Elasticsearch 索引查

询相关的知识，包括索引、映射、搜索、聚合，接着介绍 Elasticsearch 的集群、分词、重要的配置等高级功能，以及 Elasticsearch 相关的其他产品，包括告警、监控、权限管理，最后通过一个 ELK 示例结束本书。在写作的时候考虑到读者的接受能力，由浅入深地进行讲解，建议读者从前往后阅读。

本书主要包括：

第 1 章 “Elasticsearch 入门”，介绍 Elasticsearch 是什么、Apache Lucene 的基础知识、Elasticsearch 的术语、JSON 介绍、Elasticsearch 的安装运行、Elasticsearch 的 HTTP 接口和 Elasticsearch 的 Java API 接口。

第 2 章 “索引”，介绍和 Elasticsearch 索引相关的接口，包括索引管理、索引映射管理、索引别名、索引设置、索引监控、索引其他重要接口以及文档管理。

第 3 章 “映射”，介绍 Elasticsearch 文档的内部结构，Elasticsearch 支持的字段类型，除此之外，本章还将展示 Elasticsearch 内置的元字段，映射的参数和动态映射功能。

第 4 章 “搜索”，详细介绍和搜索相关的知识，包括搜索的详细参数，搜索的评分机制、滚动查询、系统内部隐藏内容的查询、搜索模板等；接着介绍 Elasticsearch 的领域查询语言 DSL (Domain-specific Language) 相关的知识点；最后介绍 Elasticsearch 的精简查询接口。

第 5 章 “聚合”，聚合可以对文档中的数据进行统计汇总、分组等，通过聚合可以完成很多的统计功能，该章介绍聚合相关的知识，包括度量聚合、分组聚合和管道聚合。

第 6 章 “集群管理”，详细介绍和集群相关的内容，包括集群的监控、集群分片迁移、集群的节点配置、集群发现、集群平衡的原理和配置。

第 7 章 “索引分词器”，介绍 Elasticsearch 的分词器和分词的原理，以及如何添加新的分词器等；还介绍 Elasticsearch 的插件相关知识，包括插件安装等。

第 8 章 “高级配置”，介绍 Elasticsearch 的高级配置，包括网络配置、脚本配置、快照和恢复配置、线程池配置和索引配置。

第 9 章 “告警、监控和权限管理”，介绍 Elasticsearch 官方支持的几个比较好的插件：Watcher、Marvel、Shield，它们可以对 Elasticsearch 进行告警、监控和权限管理。

第 10 章 “ELK 应用”，介绍 Elasticsearch 与另外两个产品 Logstash 和 Kibana 如何组合使用，Logstash 是对日志进行收集和处理，Kibana 是对存储在 Elasticsearch 中的索引进行展示和报表分析；最后通过一个简单的示例来介绍 ELK 几个产品是如何关联的。

在编写本书的时候，Elasticsearch 的最新版本是 2.2.0，但本书准备正式出版的时候，Elasticsearch 发布了最新的 5.0 版本。所以本书增加了一个附录专门介绍 5.0 版本的特性与改进。本书前面的部分截图是 2.2.0 版本的，书中所有的例子和功能都可以在 Elasticsearch 2.3.3 下运行，大部分的功能都可以在 5.0 下运行，详细的新版本差别请参考附录部分。本书中的例子大部分都是 HTTP 接口的，这些接口的测试使用了 Elasticsearch Head 插件。如果你想使用另一种工具，请注意修改 HTTP 请求的格式和编码，以便适合你所选择的工具。书中例子的结构大多是 JSON 格式，美化后的 JSON 格式比较容易阅读，但美化后的 JSON 格式比较

长，所以我们在不影响阅读的情况下，对美化后的格式做了简单调整。书中还有一小部分是 Java 接口，我们在实验时用的是 Eclipse 工具，其他主流的 Java 开发工具都适用。

本书的目标读者是对全文检索和 Elasticsearch 有兴趣的读者，如果你是一个初学者，通过本书你将学到 Elasticsearch 的基础知识，以及如何使用一些高级功能。如果你已经知道并使用了 Elasticsearch 但又想深入了解其本身，想了解如何改进查询相关性，如何使用 Elasticsearch Java API 等，也会发现本书的实用性。

由于时间紧，能力有限，编写的过程中难免有不当之处，还请各位读者不吝指出。

致谢

在此，首先我想感谢我的家庭，多年以来，因为工作关系我照顾他们太少，他们为我付出太多；我在全身心投入本书写作的时候，他们同样表现出了极大的耐心，是我最坚强的后盾。

其次还要感谢赛克蓝德公司以及公司的同事：周忠立、万荣慧和夏海华，是他们的辛勤工作才完成本书的编写工作，尤其是周忠立在很多章节的编写上贡献了很多的内容；同时要感谢本书的出版团队，尤其是吴怡编辑，写书是件非常艰巨的任务，很多内容需要反复推敲才能表达准确的意思，吴怡在检查错误、校稿、消除表达歧义等方面做出了很多贡献。

最后，非常诚挚地感谢所有 Elasticsearch 项目的创建者和开发者，感谢他们杰出的工作和对开源项目的热情。没有他们，就没有本书的诞生，没有他们，开源搜索引擎就不会有现在这种活力。再次感谢！

朱林

2016 年 10 月于南京

目录 Contents

前言	1.5.2 REST 介绍	25
	1.5.3 Head 插件安装	26
第 1 章 Elasticsearch 入门	1.5.4 创建库	27
1.1 Elasticsearch 是什么	1.5.5 插入数据	28
1.1.1 Elasticsearch 的历史	1.5.6 修改文档	28
1.1.2 相关产品	1.5.7 查询文档	29
1.2 全文搜索	1.5.8 删除文档	29
1.2.1 Lucene 介绍	1.5.9 删除库	30
1.2.2 Lucene 倒排索引	1.6 Java 接口	30
1.3 基础知识	1.6.1 Java 接口说明	30
1.3.1 Elasticsearch 术语及概念	1.6.2 创建索引文档	33
1.3.2 JSON 介绍	1.6.3 增加文档	34
1.4 安装配置	1.6.4 修改文档	35
1.4.1 安装 Java	1.6.5 查询文档	35
1.4.2 安装 Elasticsearch	1.6.6 删除文档	35
1.4.3 配置	1.7 小结	36
1.4.4 运行		
1.4.5 停止		
1.4.6 作为服务		
1.4.7 版本升级		
1.5 对外接口		
1.5.1 API 约定		
	第 2 章 索引	37
	2.1 索引管理	37
	2.1.1 创建索引	37
	2.1.2 删除索引	39
	2.1.3 获取索引	39

2.1.4 打开/关闭索引	40	3.2 字段数据类型	90
2.2 索引映射管理	41	3.2.1 核心数据类型	91
2.2.1 增加映射	41	3.2.2 复杂数据类型	96
2.2.2 获取映射	44	3.2.3 地理数据类型	100
2.2.3 获取字段映射	45	3.2.4 专门数据类型	106
2.2.4 判断类型是否存在	46	3.3 元字段	108
2.3 索引别名	46	3.3.1 _all 字段	109
2.4 索引配置	51	3.3.2 _field_names 字段	109
2.4.1 更新索引配置	51	3.3.3 _id 字段	110
2.4.2 获取配置	52	3.3.4 _index 字段	110
2.4.3 索引分析	52	3.3.5 _meta 字段	111
2.4.4 索引模板	54	3.3.6 _parent 字段	111
2.4.5 复制配置	55	3.3.7 _routing 字段	112
2.4.6 重建索引	56	3.3.8 _source 字段	114
2.5 索引监控	60	3.3.9 _type 字段	115
2.5.1 索引统计	60	3.3.10 _uid 字段	115
2.5.2 索引分片	62	3.4 映射参数	116
2.5.3 索引恢复	63	3.4.1 analyzer 参数	116
2.5.4 索引分片存储	64	3.4.2 boost 参数	118
2.6 状态管理	64	3.4.3 coerce 参数	119
2.6.1 清除缓存	64	3.4.4 copy_to 参数	120
2.6.2 索引刷新	64	3.4.5 doc_values 参数	121
2.6.3 冲洗	65	3.4.6 dynamic 参数	122
2.6.4 合并索引	65	3.4.7 enabled 参数	122
2.7 文档管理	66	3.4.8 fielddata 参数	123
2.7.1 增加文档	66	3.4.9 format 参数	126
2.7.2 更新删除文档	69	3.4.10 geohash 参数	128
2.7.3 查询文档	73	3.4.11 geohash_precision 参数	129
2.7.4 多文档操作	76	3.4.12 geohash_prefix 参数	130
2.7.5 索引词频率	80	3.4.13 ignore_above 参数	131
2.7.6 查询更新接口	83	3.4.14 ignore_malformed 参数	131
2.8 小结	87	3.4.15 include_in_all 参数	132
第3章 映射	88	3.4.16 index 参数	133
3.1 概念	88	3.4.17 index_options 参数	133
		3.4.18 lat_lon 参数	134

3.4.19	fields 参数	135
3.4.20	norms 参数	136
3.4.21	null_value 参数	137
3.4.22	position_increment_gap 参数	137
3.4.23	precision_step 参数	138
3.4.24	properties 参数	138
3.4.25	search_analyzer 参数	139
3.4.26	similarity 参数	140
3.4.27	store 参数	141
3.4.28	term_vector 参数	141
3.5	动态映射	142
3.5.1	概念	142
3.5.2	_default_ 映射	143
3.5.3	动态字段映射	143
3.5.4	动态模板	145
3.5.5	重写默认模板	148
3.6	小结	148

第4章 搜索 149

4.1	深入搜索	149
4.1.1	搜索方式	149
4.1.2	重新评分	153
4.1.3	滚动查询请求	155
4.1.4	隐藏内容查询	158
4.1.5	搜索相关函数	161
4.1.6	搜索模板	164
4.2	查询 DSL	167
4.2.1	查询和过滤的区别	167
4.2.2	全文搜索	168
4.2.3	字段查询	179
4.2.4	复合查询	183
4.2.5	连接查询	188
4.2.6	地理查询	190
4.2.7	跨度查询	197

4.2.8	高亮显示	200
4.3	简化查询	203
4.4	小结	206

第5章 聚合 207

5.1	聚合的分类	207
5.2	度量聚合	209
5.2.1	平均值聚合	209
5.2.2	基数聚合	211
5.2.3	最大值聚合	213
5.2.4	最小值聚合	214
5.2.5	和聚合	214
5.2.6	值计数聚合	215
5.2.7	统计聚合	215
5.2.8	百分比聚合	215
5.2.9	百分比分级聚合	216
5.2.10	最高命中排行聚合	217
5.2.11	脚本度量聚合	217
5.2.12	地理边界聚合	221
5.2.13	地理重心聚合	222
5.3	分组聚合	223
5.3.1	子聚合	224
5.3.2	直方图聚合	226
5.3.3	日期直方图聚合	230
5.3.4	时间范围聚合	233
5.3.5	范围聚合	234
5.3.6	过滤聚合	235
5.3.7	多重过滤聚合	236
5.3.8	空值聚合	238
5.3.9	嵌套聚合	239
5.3.10	采样聚合	240
5.3.11	重要索引词聚合	242
5.3.12	索引词聚合	245
5.3.13	总体聚合	251

5.3.14 地理点距离聚合	251
5.3.15 地理散列网格聚合	253
5.3.16 IPv4 范围聚合	255
5.4 管道聚合	257
5.4.1 平均分组聚合	259
5.4.2 移动平均聚合	261
5.4.3 总和分组聚合	262
5.4.4 总和累计聚合	262
5.4.5 最大分组聚合	264
5.4.6 最小分组聚合	265
5.4.7 统计分组聚合	266
5.4.8 百分位分组聚合	268
5.4.9 差值聚合	269
5.4.10 分组脚本聚合	273
5.4.11 串行差分聚合	275
5.4.12 分组选择器聚合	276
5.5 小结	277

第6章 集群管理 278

6.1 集群节点监控	278
6.1.1 集群健康值	278
6.1.2 集群状态	279
6.1.3 集群统计	280
6.1.4 集群任务管理	280
6.1.5 待定集群任务	281
6.1.6 节点信息	281
6.1.7 节点统计	282
6.2 集群分片迁移	283
6.3 集群节点配置	284
6.3.1 主节点	285
6.3.2 数据节点	286
6.3.3 客户端节点	286
6.3.4 部落节点	287
6.4 节点发现	287

6.4.1 主节点选举	288
6.4.2 故障检测	288
6.5 集群平衡配置	289
6.5.1 分片分配设置	289
6.5.2 基于磁盘的配置	290
6.5.3 分片智能分配	291
6.5.4 分片配置过滤	292
6.5.5 其他集群配置	293
6.6 小结	293

第7章 索引分词器 294

7.1 分词器的概念	294
7.2 中文分词器	298
7.3 插件	300
7.3.1 插件管理	301
7.3.2 插件安装	301
7.3.3 插件清单	302
7.4 小结	304

第8章 高级配置 305

8.1 网络相关配置	305
8.1.1 本地网关配置	305
8.1.2 HTTP 配置	306
8.1.3 网络配置	307
8.1.4 传输配置	308
8.2 脚本配置	310
8.2.1 脚本使用	311
8.2.2 脚本配置	313
8.3 快照和恢复配置	318
8.4 线程池配置	324
8.5 索引配置	326
8.5.1 缓存配置	326
8.5.2 索引碎片分配	329

8.5.3 合并	332
8.5.4 相似模块	332
8.5.5 响应慢日志监控	333
8.5.6 存储	335
8.5.7 事务日志	336
8.6 小结	337

第9章 告警、监控和权限管理 338

9.1 告警	338
9.1.1 安装	338
9.1.2 结构	339
9.1.3 示例	352
9.1.4 告警输出配置	354
9.1.5 告警管理	355
9.2 监控	356
9.2.1 安装	356
9.2.2 配置	357
9.3 权限管理	360
9.3.1 工作原理	361
9.3.2 用户认证	361

9.3.3 角色管理	366
9.3.4 综合示例	368
9.4 小结	369

第10章 ELK 应用 370

10.1 Logstash	370
10.1.1 配置	371
10.1.2 插件管理	374
10.2 Kibana 配置	377
10.2.1 Discover	379
10.2.2 Visualize	381
10.2.3 Dashboard	383
10.2.4 Settings	386
10.3 综合示例	387
10.4 小结	390

附录 Elasticsearch 5.0 的特性与改进 391

Elasticsearch 入门

欢迎来到 Elasticsearch 世界，它目前是全球最受欢迎的全文搜索引擎。你对 Elasticsearch 和全文搜索有没有经验都不要紧。我们希望你可以通过这本书走进 Elasticsearch 的大门。这本书是为初学者准备的，当然对于中高级的人员也有参考作用。我们首先介绍一些和 Elasticsearch 相关的基础内容。接着介绍一下 Elasticsearch 的安装和配置。此外本章还会介绍如何简单使用 Elasticsearch，包括 HTTP JSON 接口和 Java 接口。在学习这些接口的过程中，不用陷入太多的细节，后面的章节会逐步展开并细化接口内容。读完本章，你将学到以下内容：

- Elasticsearch 介绍
- 全文搜索
- Elasticsearch 的基础知识
- 安装和配置 Elasticsearch
- HTTP REST API 接口
- Java 开发接口

1.1 Elasticsearch 是什么

Elasticsearch (ES) 是一个基于 Lucene 构建的开源、分布式、RESTful 接口全文搜索引擎。Elasticsearch 还是一个分布式文档数据库，其中每个字段均是被索引的数据且可被搜索，它能够扩展至数以百计的服务器存储以及处理 PB 级的数据。它可以在很短的时间内存储、搜索和分析大量的数据。它通常作为具有复杂搜索场景情况下的核心发动机。

Elasticsearch 就是为高可用和可扩展而生的。可以通过购置性能更强的服务器来完成，

称为垂直扩展或者向上扩展 (Vertical Scale/Scaling Up), 或增加更多的服务器来完成, 称为水平扩展或者向外扩展 (Horizontal Scale/Scaling Out)。

尽管 ES 能够利用更强劲的硬件, 垂直扩展毕竟还是有它的极限。真正的可扩展性来自于水平扩展, 通过向集群中添加更多的节点来分担负载, 增加可靠性。

在大多数数据库中, 水平扩展通常都需要你对应用进行一次大的重构来利用更多的节点。而 ES 天生就是分布式的: 它知道如何管理多个节点来完成扩展和实现高可用性。这也意味着你的应用不需要做任何改动。

我们举几个例子来说明 Elasticsearch 能做什么?

当你经营一家网上商店, 你可以让你的客户搜索你卖的商品。在这种情况下, 你可以使用 Elasticsearch 来存储你的整个产品目录和库存信息, 为客户提供精准搜索, 可以为客户推荐相关商品。

当你想收集日志或者交易数据的时候, 需要分析和挖掘这些数据, 寻找趋势, 进行统计, 总结, 或发现异常。在这种情况下, 你可以使用 Logstash 或者其他工具来进行收集数据, 当这些数据存储到 Elasticsearch 中。你可以搜索和汇总这些数据, 找到任何你感兴趣的信息。

当你运行一个价格提醒的平台, 可以给客户提供一些规则, 例如客户有兴趣购买一个电子设备, 当商品的价格在未来一个月内价格低于多少钱的时候通知客户。在这种情况下, 你可以把供应商的价格, 把他们定期存储到 Elasticsearch 中, 使用定时器过滤来匹配客户的需求, 当查询到价格低于客户设定的值后给客户发送一条通知。

当有大量数据 (千万条以上的记录) 时, 你有商业智能分析的需求, 希望快速调查、分析和可视化。在这种情况下, 你可以使用 Elasticsearch 来存储你的数据, 然后用 Kibana 建立自定义的仪表板或者任何你熟悉的语言开发展示界面, 你可以使用 Elasticsearch 的聚合功能来执行复杂的商业智能与数据查询。

对于程序员来说, 比较有名的案例是 GitHub, GitHub 的搜索是基于 Elasticsearch 构建的, 在 github.com/search 页面, 你可以搜索项目、用户、issue、pull request, 还有代码。共有 40 ~ 50 个索引库, 分别用于索引网站需要跟踪的各种数据。虽然只索引项目的主分支 (master), 但这个数据量依然巨大, 包括 20 亿个索引文档, 30TB 的索引文件。

1.1.1 Elasticsearch 的历史

网上流传的故事是: 多年前, 一个名叫 Shay Banon 的刚结婚不久的失业开发者, 由于妻子要去伦敦学习厨师, 他便跟着也去了。在他找工作的过程中, 为了给妻子构建一个食谱的搜索引擎, 他开始构建一个早期版本的 Lucene。

直接基于 Lucene 工作会比较困难, 所以 Shay 开始抽象 Lucene 代码以便 Java 程序员可以在应用中添加搜索功能。他发布了第一个开源项目, 叫作 “Compass”。

后来 Shay 找到一份工作, 这份工作处在高性能和内存数据网格的分布式环境中, 因此

高性能的、实时的、分布式的搜索引擎也是理所当然需要的。然后他决定重写 Compass 库使其成为一个独立的服务叫作 Elasticsearch。

第一个公开版本发布于 2010 年 2 月，在那之后 Elasticsearch 已经成为 GitHub 上最受欢迎的项目之一，代码贡献者超过 300 人。直到 2016 年 3 月 30 日，Elasticsearch 已经发布了 2.3.0 版本。目前已经成为全球最受欢迎的全文搜索引擎。

那 Elasticsearch 为什么会有如此的魅力呢？我们首先看一下 Elasticsearch 的优点：

- **横向可扩展性**：只需要增加一台服务器，做一点儿配置，启动一下 Elasticsearch 进程就可以并入集群。
- **分片机制提供更好的分布性**：同一个索引分成多个分片（sharding），这点类似于 HDFS 的块机制；分而治之的方式可提升处理效率。
- **高可用**：提供复制（replica）机制，一个分片可以设置多个复制，使得某台服务器在宕机的情况下，集群仍旧可以照常运行，并会把服务器宕机丢失的数据信息复制恢复到其他可用节点上。
- **使用简单**：只需一条命令就可以下载文件，然后很快就能搭建一个站内搜索引擎。

1.1.2 相关产品

Beats：它是一个代理，将不同类型的数据发送到 Elasticsearch 中。它可以直接将数据发送到 Elasticsearch。Beats 由三部分内容组成：Filebeat、Topbeat、Packetbeat。Filebeat 用来收集日志。Topbeat 用来收集系统基础设置数据，如 CPU、内存、每个进程的统计信息。Packetbeat 是一个网络包分析工具，统计收集网络信息。这三个工具是官方提供的。

Shield：它为 Elasticsearch 带来企业级的安全性，加密通信，认证保护整个 Elasticsearch 数据，它是基于角色的访问控制与审计。当今企业对安全需求越来越重视，Shield 可以提供安全的 Elasticsearch 访问，从而保护核心的数据。注意：Shield 是收费的产品。

Watcher：它是 Elasticsearch 的警报和通知工具。它可以主动监测 Elasticsearch 的状态，并在有异常的时候进行提醒，还可以根据你的数据变化情况来采取不同的处理方式。注意：Watcher 也是收费的产品。

Marvel：它是 Elasticsearch 的管理和监控工具。它监测 Elasticsearch 集群索引和节点的活动，快速诊断问题。注意：Marvel 也是收费的产品。

1.2 全文搜索

全文搜索是指计算机搜索程序通过扫描文章中的每一个词，对每一个词建立一个索引，指明该词在文章中出现的次数和位置，当用户查询时，搜索程序就根据事先建立的索引进行查找，并将查找的结果反馈给用户。这个过程类似于通过字典中的搜索字表查字的过程。Lucene 是目前全球使用最广的全文搜索引擎开源库。

1.2.1 Lucene 介绍

Lucene 是 Apache 软件基金会中一个开放源代码的全文搜索引擎工具包，是一个全文搜索引擎的架构，提供了完整的查询引擎和索引引擎，部分文本分析引擎。Lucene 的目的是为软件开发人员提供一个简单易用的工具包，以方便在目标系统中实现全文检索的功能，或者是以此为基础建立起完整的全文搜索引擎。

Lucene 最初是由 Doug Cutting 所撰写的，是一位资深全文索引/搜索专家，曾经是 V-Twin 搜索引擎的主要开发者，后来在 Excite 担任高级系统架构设计师，目前从事于一些 Internet 底层架构的研究。

1.2.2 Lucene 倒排索引

倒排索引源于实际应用中需要根据属性的值来查找记录。这种索引表中的每一项都包括一个属性值和具有该属性值的各记录的地址。由于不是由记录来确定属性值，而是由属性值来确定记录的位置，因而称为倒排索引（inverted index）。带有倒排索引的文件我们称为倒排索引文件，简称倒排文件（inverted file）。

倒排索引中的索引对象是文档或者文档集中的单词等，用来存储这些单词在一个文档或者一组文档中的存储位置，是对文档或者文档集合的一种最常用的索引机制。

搜索引擎的关键步骤就是建立倒排索引，倒排索引一般表示为一个关键词，然后是它的频度（出现的次数）、位置（出现在哪一篇文章或网页中，及有关的日期，作者等信息），好比一本书的目录、标签一般。读者想看哪一个主题相关的章节，直接根据目录即可找到相关的页面。不必再从书的第一页到最后一页，一页一页地查找。

Lucene 使用的是倒排文件索引结构，下面用例子介绍该结构及相应的生成算法。

假设有两篇文章 1 和文章 2。

文章 1 的内容为：Tom lives in Guangzhou,I live in Guangzhou too.

文章 2 的内容为：He once lived in Shanghai.

1. 取得关键词

由于 Lucene 是基于关键词索引和查询的，首先我们要取得这两篇文章的关键词，通常我们需要如下处理措施：

- ❑ 我们现在有的是文章内容，即一个字符串，我们先要找出字符串中的所有单词，即分词。英文单词由于用空格分隔，比较好处理。中文单词间由于是连在一起的，所以需要特殊的分词处理。
- ❑ 文章中的“in”“once”“too”等词没有什么实际意义，中文中的“的”“是”等字通常也无具体含义，这些不代表概念的词是可以过滤掉的。
- ❑ 用户通常希望查“He”时能把含“he”和“HE”的文章也找出来，所以所有单词需要统一大小写。

□ 用户通常希望查“live”时能把含“lives”和“lived”的文章也找出来，所以需要把“lives”，“lived”还原成“live”。

□ 文章中的标点符号通常不表示某种概念，也可以过滤掉。

在 Lucene 中以上措施由 Analyzer 类完成。经过上面处理后，得到如下结果：

文章 1 的所有关键词为：[tom] [live] [guangzhou] [i] [live] [guangzhou]

文章 2 的所有关键词为：[he] [live] [shanghai]

2. 建立倒排索引

有了关键词后，我们就可以建立倒排索引了。上面的对应关系是：“文章号”对“文章中所有关键词”。倒排索引把这个关系倒过来，变成：“关键词”对“拥有该关键词的所有文章号”。

文章 1 和文章 2 经过倒排后的对应关系见表 1-1。

表 1-1 倒排索引关键词文章号对应关系示例

关键词	文章号
guangzhou	1
he	2
i	1
live	1, 2
shanghai	2
tom	1

通常仅知道关键词在哪些文章中出现还

不够，我们还需要知道关键词在文章中出现的次数和位置，通常有两种位置：

□ 字符位置，即记录该词是文章中第几个字符（优点是显示并定位关键词快）。

□ 关键词位置，即记录该词是文章中第几个关键词（优点是节约索引空间、词组查询快），Lucene 中记录的就是这种位置。

加上“出现频率”和“出现位置”信息后，我们的索引结构参见表 1-2。

表 1-2 倒排索引关键词频率位置示例

关键词	文章号 [出现频率]	出现位置
guangzhou	1[2]	3,6
he	2[1]	1
i	1[1]	4
live	1[2]	2,5
	2[1]	2
shanghai	2[1]	3
tom	1[1]	1

以 live 这行为例，我们说明一下该结构：live 在文章 1 中出现了 2 次，文章 2 中出现了一次，它的出现位置为“2,5,2”这表示什么呢？我们需要结合文章号和出现频率来分析，文章 1 中出现了 2 次，那么“2,5”就表示 live 在文章 1 中出现的两个位置，文章 2 中出现了一次，剩下的“2”就表示 live 是文章 2 中的第 2 个关键字。

以上就是 Lucene 索引结构中最核心的部分。我们注意到关键字是按字符顺序排列的

(Lucene 没有使用 B 树结构), 因此 Lucene 可以用二元搜索算法快速定位关键词。

3. 实现

实现时, Lucene 将上面三列分别作为词典文件 (Term Dictionary)、频率文件 (frequencies)、位置文件 (positions) 保存。其中词典文件不仅保存了每个关键词, 还保留了指向频率文件和位置文件的指针, 通过指针可以找到该关键字的频率信息和位置信息。

Lucene 中使用了 field 的概念, 用于表达信息所在位置 (如标题中、文章中、URL 中), 在建索引中, 该 field 信息也记录在词典文件中, 每个关键词都有一个 field 信息, 因为每个关键字一定属于一个或多个 field。

4. 压缩算法

为了减小索引文件的大小, Lucene 对索引还使用了压缩技术。

首先, 对词典文件中的关键词进行了压缩, 关键词压缩为 <前缀长度, 后缀>, 例如: 当前词为“阿拉伯语”, 上一个词为“阿拉伯”, 那么“阿拉伯语”压缩为 <3, 语>。

其次大量用到的是对数字的压缩, 数字只保存与上一个值的差值 (这样可以减少数字的长度, 进而减少保存该数字需要的字节数)。例如当前文章号是 16389 (不压缩要用 3 个字节保存), 上一篇文章号是 16382, 压缩后保存 7 (只用一个字节)。

5. 应用场景

下面我们可以通过对该索引的查询来解释一下为什么要建立索引。

假设要查询单词“live”, Lucene 先对词典二元查找、找到该词, 通过指向频率文件的指针读出所有文章号, 然后返回结果。词典通常非常小, 因而, 整个过程的时间是毫秒级的。

而用普通的顺序匹配算法, 不建索引, 而是对所有文章的内容进行字符串匹配, 这个过程将会相当缓慢, 当文章数目很大时, 时间往往是无法忍受的。

1.3 基础知识

在 Elasticsearch 中有很多的术语和概念, 为了后面更好地理解和阅读本书, 本节先介绍一下这些术语和概念, 然后介绍一下 Elasticsearch 存储的格式 JSON。

1.3.1 Elasticsearch 术语及概念

1. 索引词 (term)

在 Elasticsearch 中索引词 (term) 是一个能够被索引的精确值。foo、Foo、FOO 几个单词是不同的索引词。索引词 (term) 是可以通过 term 查询进行准确的搜索。

2. 文本 (text)

文本是一段普通的非结构化文字。通常，文本会被分析成一个个的索引词，存储在 Elasticsearch 的索引库中。为了让文本能够进行搜索，文本字段需要事先进行分析；当对文本中的关键词进行查询的时候，搜索引擎应该根据搜索条件搜索出原文本。

3. 分析 (analysis)

分析是将文本转换为索引词的过程，分析的结果依赖于分词器。比如：FOO BAR、Foo-Bar 和 foo bar 这几个单词有可能会被分析成相同的索引词 foo 和 bar，这些索引词存储在 Elasticsearch 的索引库中。当用 FoO:bAR 进行全文搜索的时候，搜索引擎根据匹配计算也能在索引库中搜索出之前的内容。这就是 Elasticsearch 的搜索分析。

4. 集群 (cluster)

集群由一个或多个节点组成，对外提供服务，对外提供索引和搜索功能。在所有节点，一个集群有一个唯一的名称默认为“Elasticsearch”。此名称是很重要的，因为每个节点只能是集群的一部分，当该节点被设置为相同的集群名称时，就会自动加入集群。当需要有多个集群的时候，要确保每个集群的名称不能重复，否则，节点可能会加入错误的集群。请注意，一个节点只能加入一个集群。此外，你还可以拥有多个独立的集群，每个集群都有其不同的集群名称。例如，在开发过程中，你可以建立开发集群库和测试集群库，分别为开发、测试服务。Elasticsearch 集群结构见图 1-1。

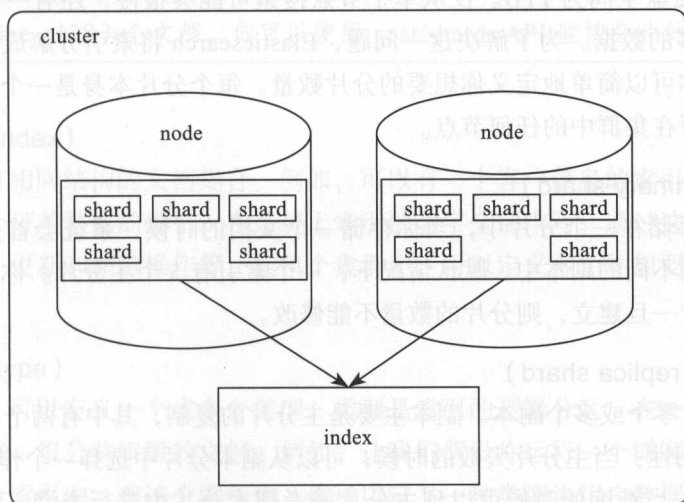


图 1-1 Elasticsearch 集群结构

5. 节点 (node)

一个节点是一个逻辑上独立的服务，它是集群的一部分，可以存储数据，并参与集群的索引和搜索功能。就像集群一样，节点也有唯一的名字，在启动的时候分配。如果你不

想要默认名称，你可以定义任何你想要的节点名。这个名字在管理中很重要，在网络中 Elasticsearch 集群通过节点名称进行管理和通信。一个节点可以被配置加入一个特定的集群。默认情况下，每个节点会加入名为 Elasticsearch 的集群中，这意味着如果你在网络上启动多个节点，如果网络畅通，他们能彼此发现并自动加入一个名为 Elasticsearch 的集群中。在一个集群中，你可以拥有多个你想要的节点。当网络没有集群运行的时候，只要启动任何一个节点，这个节点会默认生成一个新的集群，这个集群会有一个节点。

6. 路由 (routing)

当存储一个文档的时候，它会存储在唯一的主分片中，具体哪个分片是通过散列值进行选择。默认情况下，这个值是由文档的 ID 生成。如果文档有一个指定的父文档，则从父文档 ID 中生成，该值可以在存储文档的时候进行修改。

7. 分片 (shard)

分片是单个 Lucene 实例，这是 Elasticsearch 管理的比较底层的功能。索引是指向主分片和副本分片的逻辑空间。对于使用，只需要指定分片的数量，其他不需要做过多的事情。在开发使用的过程中，我们对应的对象都是索引，Elasticsearch 会自动管理集群中所有的分片，当发生故障的时候，Elasticsearch 会把分片移动到不同的节点或者添加新的节点。

一个索引可以存储很大的数据，这些空间可以超过一个节点的物理存储的限制。例如，十亿个文档占用磁盘空间为 1TB。仅从单个节点搜索可能会很慢，还有一台物理机器也不一定能存储这么多的数据。为了解决这一问题，Elasticsearch 将索引分解成多个分片。当你创建一个索引，你可以简单地定义你想要的分片数量。每个分片本身是一个全功能的、独立的单元，可以托管在集群中的任何节点。

8. 主分片 (primary shard)

每个文档都存储在一个分片中，当你存储一个文档的时候，系统会首先存储在主分片中，然后会复制到不同的副本中。默认情况下，一个索引有 5 个主分片。你可以事先制定分片的数量，当分片一旦建立，则分片的数量不能修改。

9. 副本分片 (replica shard)

每一个分片有零个或多个副本。副本主要是主分片的复制，其中有两个目的：

- ☐ 增加高可用性：当主分片失败的时候，可以从副本分片选择一个作为主分片。
- ☐ 提高性能：当查询的时候可以到主分片或者副本分片中进行查询。默认情况下，一个主分片配有一个副本，但副本的数量可以在后面动态地配置增加。副本分片必须部署在不同的节点上，不能部署在和主分片相同的节点上。

分片主要有两个很重要的原因是：

- ☐ 允许水平分割扩展数据。

□ 允许分配和并行操作（可能在多个节点上）从而提高性能和吞吐量。

这些很强大的功能对用户来说是透明的，你不需要做什么操作，系统会自动处理。

10. 复制 (replica)

复制是一个非常有用的功能，不然会有单点问题。当网络中的某个节点出现问题的时候，复制可以对故障进行转移，保证系统的高可用。因此，Elasticsearch 允许你创建一个或多个拷贝，你的索引分片就形成了所谓的副本或副本分片。

复制是重要的，主要的原因有：

□ 它提供了高可用性，当节点失败的时候不受影响。需要注意的是，一个复制的分片不会存储在同一个节点中。

□ 它允许你扩展搜索量，提高并发量，因为搜索可以在所有副本上并行执行。

每个索引可以拆分成多个分片。索引可以复制零个或者多个分片。一旦复制，每个索引就有了主分片和副本分片。分片的数量和副本的数量可以在创建索引时定义。当创建索引后，你可以随时改变副本的数量，但你不能改变分片的数量。

默认情况下，每个索引分配 5 个分片和一个副本，这意味着你的集群节点至少要有两个节点，你将拥有 5 个主要的分片和 5 个副本分片共计 10 个分片。



注意 每个 Elasticsearch 分片是一个 Lucene 的索引。有文档存储数量限制，你可以在一个单一的 Lucene 索引中存储的最大值为 `lucene-5843`，极限是 `2147483519 (= integer.max_value - 128)` 个文档。你可以使用 `_cat/shards` API 监控分片的大小。

11. 索引 (index)

索引是具有相同结构的文档集合。例如，可以有一个客户信息的索引，包括一个产品目录的索引，一个订单数据的索引。在系统上索引的名字全部小写，通过这个名字可以用来执行索引、搜索、更新和删除操作等。在单个集群中，可以定义多个你想要的索引。索引结构参见图 1-2。

12. 类型 (type)

在索引中，可以定义一个或多个类型，类型是索引的逻辑分区。在一般情况下，一种类型被定义为具有一组公共字段的文档。例如，让我们假设你运行一个博客平台，并把所有的数据存储在一个索引中。在这个索引中，你可以定义一种类型为用户数据，一种类型为博客数据，另一种类型为评论数据。

13. 文档 (document)

文档是存储在 Elasticsearch 中的一个 JSON 格式的字符串。它就像在关系数据库中表的一行。每个存储在索引中的一个文档都有一个类型和一个 ID，每个文档都是一个 JSON 对

象，存储了零个或者多个字段，或者键值对。原始的 JSON 文档被存储在一个叫作 `_source` 的字段中。当搜索文档的时候默认返回的就是这个字段。

index				
type				
document	field	field	field	field
document	field	field	field	field
document	field	field	field	field
type				
document	field	field	field	field
document	field	field	field	field
document	field	field	field	field

图 1-2 Elasticsearch 索引术语结构

14. 映射 (mapping)

映射像关系数据库中的表结构，每一个索引都有一个映射，它定义了索引中的每一个字段类型，以及一个索引范围内的设置。一个映射可以事先被定义，或者在第一次存储文档的时候自动识别。

15. 字段 (field)

文档中包含零个或者多个字段，字段可以是一个简单的值（例如字符串、整数、日期），也可以是一个数组或对象的嵌套结构。字段类似于关系数据库中表的列。每个字段都对应一个字段类型，例如整数、字符串、对象等。字段还可以指定如何分析该字段的值。

16. 来源字段 (source field)

默认情况下，你的原文档将被存储在 `_source` 这个字段中，当你查询的时候也是返回这个字段。这允许你可以从搜索结果中访问原始的对象，这个对象返回一个精确的 JSON 字符串，这个对象不显示索引分析后的其他任何数据。

17. 主键 (ID)

ID 是一个文件的唯一标识，如果在存库的时候没有提供 ID，系统会自动生成一个 ID，文档的 `index/type/id` 必须是唯一的。

1.3.2 JSON 介绍

在 Elasticsearch 中的接口中，大多数都是以 JSON 的格式进行的，那 JSON 是什么呢？

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人们阅读和编写,同时也易于机器解析和生成。它基于 JavaScript Programming Language, Standard ECMA-262 3rd Edition - December 1999 的一个子集。JSON 采用完全独立于语言的文本格式,但是也使用了类似于 C 语言家族的习惯 (包括 C、C++、C#、Java、JavaScript、Perl、Python 等)。这些特性使 JSON 成为理想的数据交换格式。

JSON 有两种结构:

□ “名称 / 值” 对的集合 (A collection of name/value pairs)。不同的语言中的理解不同,如对象 (object)、纪录 (record)、结构 (struct)、字典 (dictionary)、哈希表 (hash table)、有键列表 (keyed list) 或者关联数组 (associative array)。

□ 值的有序列表 (An ordered list of values)。在大部分语言中,是指数组 (array)。

这些都是常见的数据结构。事实上大部分现代计算机语言都以某种形式支持它们。这使得一种数据格式在同样基于这些格式的编程语言之间进行交换成为可能。

例 1-1

1. “名称 / 值” 对

按照最简单的形式,可以用下面这样的 JSON 表示 “名称 / 值” 对:

```
{"firstName": "Brett"}
```

这个示例非常基础,而且实际上比等效的纯文本 “名称 / 值” 对占用更多的空间:

```
firstName=Brett
```

但是,当将多个 “名称 / 值” 对串在一起时,JSON 就会体现出它的价值了。首先,可以创建包含多个 “名称 / 值” 对的记录,比如:

```
{"firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa"}
```

从语法方面来看,这与 “名称 / 值” 对相比并没有很大的优势,但是在这种情况下 JSON 更容易使用,而且可读性更好。例如,它明确地表示以上三个值都是同一记录的一部分;用花括号将这些值进行关联。

2. 表示数组

当需要表示一组值时,JSON 不但能够提高可读性,而且可以减少复杂性。例如,假设你希望表示一个人名列表。在 XML 中,需要许多开始标记和结束标记;如果使用典型的 “名称 / 值” 对,那么必须建立一种专有的数据格式,或者将键名称修改为 person1-firstName 这样的形式。

如果使用 JSON,就只需将多个带花括号的记录分组在一起:

```
{
  "people": [
```

```
{ "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" },
{ "firstName": "Jason", "lastName": "Hunter", "email": "bbbb" },
{ "firstName": "Elliotte", "lastName": "Harold", "email": "cccc" }
```

```
]
```

这不难理解。在这个示例中，只有一个名为 `people` 的变量，值是包含三个条目的数组，每个条目是一个人的记录，其中包含名、姓和电子邮件地址。上面的示例演示如何用括号将记录组合成一个值。当然，可以使用相同的语法表示多个值（每个值包含多个记录），如下所示：

```
{
  "programmers": [{ "firstName": "Brett", "lastName": "McLaughlin", "email": "aaaa" },
                  { "firstName": "Jason", "lastName": "Hunter", "email": "bbbb" } ],
  "authors": [{ "firstName": "Isaac", "lastName": "Asimov", "genre": "sciencefiction" },
               { "firstName": "Sergei", "lastName": "Rachmaninoff", "instrument": "piano" } ]
}
```

这里最值得注意的是，能够表示多个值，每个值进而包含多个值。但是还应该注意，在不同的主条目（`programmers`、`authors` 和 `musicians`）之间，记录中实际的“名称 / 值”对可以不同。JSON 是完全动态的，允许在 JSON 结构的中间改变表示数据的方式。

在处理 JSON 格式的数据时，没有需要遵守的预定义的约束。所以，在同样的数据结构中，可以改变表示数据的方式，甚至可以用不同方式表示同一事物。

1.4 安装配置

下面是安装 Elasticsearch 所需要的几个步骤，接下来几节将详细说明。

1.4.1 安装 Java

为了运行 Elasticsearch，第一步是要安装 Java，Elasticsearch 需要 Java 7 或者更高版本的支持。建议使用最新的 Oracle 的 JDK 版本 1.8.0_72。如需了解 Java 的情况，可以到 Oracle 的官网找相关的资料。在你安装 Elasticsearch 前，请检查你的 Java 版本运行：

```
java -version
```

1.4.2 安装 Elasticsearch

当我们设置好 Java 后，下载最新的 Elasticsearch 版本，解压，安装完毕。下载地址为：<https://www.elastic.co/downloads/past-releases/elasticsearch-2-3-0>。

Elasticsearch 是 Java 开发的，所以 JVM 的环境变量 `JAVA_OPTS` 对 Elasticsearch 是非常重要的。在 `JAVA_OPTS` 中对 Elasticsearch 最重要的参数是 `-Xmx` 最大可以使用内存的参数，一般情况下大内存更能发挥 Elasticsearch 作用，建议 `-Xmx` 设置为物理内存的一半，为

了减少内存分配带来的性能损耗，最好一开始就设置初始内存和最大内存都为物理内存的一半，即 Xms 和 Xmx 这两个参数。

由于 JAVA_OPTS 大多数时候对整个机器环境起作用，所以最好是保留默认的 JAVA_OPTS，最好用 ES_JAVA_OPTS 环境变量设置来作为 JAVA_OPTS 参数：

默认的配置文件在 elasticsearch/bin/elasticsearch.in.sh 中。

```
if [ "x$ES_MIN_MEM" = "x" ]; then
    ES_MIN_MEM=256m
fi
if [ "x$ES_MAX_MEM" = "x" ]; then
    ES_MAX_MEM=1g
fi
if [ "x$ES_HEAP_SIZE" != "x" ]; then
    ES_MIN_MEM=$ES_HEAP_SIZE
    ES_MAX_MEM=$ES_HEAP_SIZE
fi

JAVA_OPTS="$JAVA_OPTS -Xms${ES_MIN_MEM}"
JAVA_OPTS="$JAVA_OPTS -Xmx${ES_MAX_MEM}"
```

ES_HEAP_SIZE 环境变量允许设置被分配到 Elasticsearch java 进程中堆内存的大小。最小值和最大值将分配相同的值，可以通过设置 ES_MIN_MEM（默认为 256M）和 ES_MAX_MEM（默认为 1G）对堆内存进行设置。

1.4.3 配置

Elasticsearch 配置文件在 elasticsearch/config 文件夹下。在这个文件夹中有两个文件，一个是 Elasticsearch 配置不同模块的配置文件 elasticsearch.yml，另一个是 Elasticsearch 日志的配置文件 logging.yml。默认配置文件的格式为 YML。

Elasticsearch 提供了多种方式进行设置，在系统内部，都使用命名空间来表示这些设置，根据这些命名空间，系统可以很容易地扩展到其他格式。比如我们设置节点名称如下：

1) yml 格式为：node.name: node-1。

2) JSON 格式为：只需要把 elasticsearch.yml 名修改成 elasticsearch.json。

配置的方式为：

```
{
  "node" : { "name" : "node-1" }
}
```

3) 通过 Elasticsearch 命令的参数来设置配置信息。例如：

```
elasticsearch -Des.node.name=node-1
```

4) Elasticsearch 还可以通过交互式方式进行设置，通过 \${prompt.text} 或者 \${prompt.secret} 来指定，\${prompt.secret} 表示在终端中隐藏输入的值，\${prompt.text} 表示在终端中

显示输入的值，例如：

```
node.name: ${prompt.text}
```

在 Elasticsearch 命令执行时，将提示你输入的实际值，例如下面的提示：

```
Enter value for [node.name]:
```



注意 当 Elasticsearch 作为服务或者在后台启动的时候，这两个参数不起作用。

1. elasticsearch.yml 配置说明

集群名称：

```
cluster.name: my-application
```

确保在不同的环境中集群的名称不重复，否则，节点可能会连接到错误的集群上。

节点名称：

```
node.name: node-1
```

默认情况下，当节点启动时 Elasticsearch 将随机在一份 3000 个名字的列表中随机指定一个。如果机器上只运行一个集群 Elasticsearch 节点，可以用 `$ {HOSTNAME}` 设置节点的名称为主机名。

节点描述：

```
node.rack: r1
```

索引存储位置：

```
path.data: /path/to/data
```

日志存储位置：

```
path.logs: /path/to/logs
```

内存分配模式：

```
bootstrap.mlockall: true
```

绑定的网卡 IP：

```
network.host: 192.168.0.1
```

http 协议端口：

```
http.port: 9200
```

开始发现新节点的 IP 地址：

```
discovery.zen.ping.unicast.hosts: ["host1", "host2"]
```

最多发现主节点的个数：

```
discovery.zen.minimum_master_nodes: 3
```

当重启集群节点后最少启动 N 个节点后开始做恢复:

```
gateway.recover_after_nodes: 3
```

在一台机器上最多启动的节点数:

```
node.max_local_storage_nodes: 1
```

当删除一个索引的时候, 需要指定具体索引的名称:

```
action.destructive_requires_name: true
```

2. 索引配置说明

在集群中创建的索引可以提供每个索引自己的设置。例如, 下面创建一个索引刷新间隔是 5 秒钟而不是默认的刷新间隔 (格式可以是 YAML 或 JSON):

请求: PUT http://127.0.0.1:9200/kimchy

参数: index:refresh_interval: 5s

这个索引参数可以设置在节点上, 例如, 在 elasticsearch.yml 文件中可以设置:

```
index.refresh_interval: 5s
```

这意味着除非索引明确定义, 这个节点上创建的每个索引的刷新间隔为 5 秒。

当然也可以在启动 Elasticsearch 的时候用参数指定:

```
elasticsearch -Des.index.refresh_interval=5s
```

3. 日志配置说明

Elasticsearch 内部使用 log4j 记录系统日志, 它试图通过使用 YAML 配置方式来简化 log4j 的配置, 配置文件位置为 elasticsearch/config/ logging.yml。JSON 格式和键值对的格式也是支持的。可以加载多个配置文件, 在启动 Elasticsearch 后系统自动合并多个配置文件。支持不同的后缀格式, 例如: (.yml, .yaml, .json or .properties)。

记录器部分包含 java 包和相应的日志级别, 在配置里可以省略 org.elasticsearch 前缀。Appender 部分包含日志描述信息。更多内容请参见 log4j。

由于日志比较重要, 正常情况下不要禁止日志的产生, 如果感觉日志过多, 可以提高日志的级别。系统日志每日会生成一个新的文件。当遇到问题的时候, 首先要检查一下日志文件, 看看是否有出错的信息。

1.4.4 运行

在 Windows 下执行 elasticsearch.bat, 在 Linux 下运行 ./elasticsearch。

如果一切顺利的话, 你应该看到如下信息:

```
[2016-02-03 16:53:31,122][INFO ][node                ] [Rintrah] version[2.2.0],
```

```

pid[6840], build[8ff36d1/2016-01-27T13:32:39Z]
[2016-02-03 16:53:31,122][INFO ][node                ] [Rintrah] initializing ...
[2016-02-03 16:53:31,668][INFO ][plugins          ] [Rintrah] modules [lang-
groovy, lang-expression], plugins [], sites []
[2016-02-03 16:53:31,684][INFO ][env                ] [Rintrah] using [1] data
paths, mounts [[work (D:)]], net usable_space [67.2gb], net total_space
[99.9gb], spins? [unknown], types [NTFS]
[2016-02-03 16:53:31,684][INFO ][env                ] [Rintrah] heap size
[910.5mb], compressed ordinary object pointers [true]
[2016-02-03 16:53:33,637][INFO ][node                ] [Rintrah] initialized
[2016-02-03 16:53:33,637][INFO ][node                ] [Rintrah] starting ...
[2016-02-03 16:53:33,918][INFO ][transport          ] [Rintrah] publish_address
{127.0.0.1:9300}, bound_addresses {[::1]:9300}, {127.0.0.1:9300}
[2016-02-03 16:53:33,934][INFO ][discovery          ] [Rintrah] elasticsearch/
1oo5dtelT8ax-3LmnTrs8g
[2016-02-03 16:53:37,982][INFO ][cluster.service    ] [Rintrah] new_master
{Rintrah}{1oo5dtelT8ax-3LmnTrs8g}{127.0.0.1}{127.0.0.1:9300}, reason: zen-
disco-join(elected_as_master, [0] joins received)
[2016-02-03 16:53:40,363][INFO ][gateway            ] [Rintrah] recovered [0]
indices into cluster_state
[2016-02-03 16:53:40,567][INFO ][http               ] [Rintrah] publish_address
{127.0.0.1:9200}, bound_addresses {[::1]:9200}, {127.0.0.1:9200}
[2016-02-03 16:53:40,567][INFO ][node                ] [Rintrah] started

```

这样我们就已经启动了 Elasticsearch，当然我们也可以在启动的时候修改集群的名称和节点的名称。例如：

```
./elasticsearch --cluster.name my_cluster_name --node.name my_node_name
```

默认情况下，Elasticsearch 使用 9200 端口提供的 REST API。该端口是可配置的。

在本机访问 <http://127.0.0.1:9200/>。

将会得到以下内容：

```

{
  "name" : "Rintrah",
  "cluster_name" : "elasticsearch",
  "version" : { "number" : "2.2.0",
    "build_hash" : "8ff36d139e16f8720f2947ef62c8167a888992fe",
    "build_timestamp" : "2016-01-27T13:32:39Z", "build_snapshot" : false,
    "lucene_version" : "5.4.1"
  },
  "tagline" : "You Know, for Search"
}

```

现在，我们的节点（和集群）将正确运行，下一步就是要了解如何进行使用。Elasticsearch 提供了非常全面和强大的 REST API，通过这些 API，我们可以了解集群的信息。这些 API 可以做如下事情：

- 1) 检查集群、节点和索引的情况、状态和统计。
- 2) 管理集群、节点、索引数据和文档数据。

3) 执行 CRUD (创建、读取、更新和删除) 操作, 可以对索引进行操作。

4) 执行高级搜索操作, 如分页、排序、过滤、脚本、聚合及其他操作。

1.4.5 停止

如果需要停止 Elasticsearch, 有以下两种方法。

❑ 如果节点是连接到控制台, 按下 Ctrl+C。

❑ 杀进程 (Linux 是 kill, Windows 下用任务管理器)。

由于 Elasticsearch 常用在集群中, 集群停止相对有些复杂, 参考 1.4.7 版本升级。

1.4.6 作为服务

1. Linux 下作为服务

Elasticsearch 创建了 debian 安装包和 RMP 安装包, 可以在官网的下载页面中进行下载。安装包需要依赖 Java, 除此就没有任何依赖。

在 Debian 系统下可以使用标准的系统工具, init 脚本放在 /etc/init.d/elasticsearch 下, 配置文件默认放在 /etc/default/elasticsearch 下。从 debian 软件包安装好后默认是不启动服务的。其原因是为了防止实例不小心加入集群。安装好后用 dpkg -i 命令来确保, 当系统启动后启动 Elasticsearch 需要运行下面的两个命令:

```
sudo update-rc.d elasticsearch defaults 95 10
sudo /etc/init.d/elasticsearch start
```

当用户运行 Debian8 或者 Ubuntu14 或者更高版本的时候, 系统需要用 systemd 来代替 update-rc.d, 在这种情况下, 请使用 systemd 来运行, 参见下面的介绍。

Elasticsearch 通常的建议是使用 Oracle 的 JDK。可以用下面的命令安装:

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
java -version
```

基于 RPM 的系统一般使用 chkconfig 来启用和禁用服务。init 脚本位于 /etc/init.d/elasticsearch 下, 配置文件放在 /etc/sysconfig/Elasticsearch 下。同 Debian 系统类似, 安装好后也不会自动加入自启动服务中。需要手工指定:

```
sudo /sbin/chkconfig --add elasticsearch
sudo service elasticsearch start
```

2. systemd 服务启动

很多 Linux 系统, 例如 Debian Jessie、Ubuntu 14 等, 系统不使用 chkconfig 来注册服务, 取而代之的是用 systemd 来启动和停止服务。命令是 /bin/systemctl 来启动和停止服务。RPM 包安装的配置文件在 /etc/sysconfig/elasticsearch 下, deb 包安装的配置文件在 /etc/

default/elasticsearch 下。安装 RPM 之后，你必须改变系统配置，然后启动 Elasticsearch。

```
sudo /bin/systemctl daemon-reload
sudo /bin/systemctl enable elasticsearch.service
sudo /bin/systemctl start elasticsearch.service
```

同时注意改变在 /etc/sysconfig/elasticsearch 中的 MAX_MAP_COUNT 设置是没有任何效果的。需要改变 /usr/lib/sysctl.d/elasticsearch.conf 中的配置才起作用。

3. Windows 下作为服务

Windows 用户可以配置 Elasticsearch 作为服务在后台运行，或在没有任何用户交互启动时自动启动。这可以通过 bin 目录下的 service.bat 脚本来实现，可以安装、卸载、管理或配置服务命令行为：service.bat install|remove|start|stop|manager [SERVICE_ID]

SERVICE_ID 是服务 ID，可以不用指定用默认的值，系统可以安装多个服务。Manager 是启动图形界面的配置。例如运行 service install 后显示的内容如下：

```
Installing service      : "elasticsearch-service-x64"
Using JAVA_HOME (64-bit): "C:\Program Files\Java\jdk1.7.0_79"
The service 'elasticsearch-service-x64' has been installed.
```

安装好后，有两种方法可以对服务进行设置。

- ❑ 图形化界面，可以用命令 service manager 来启动图形界面。执行后显示界面如图 1-3 所示。

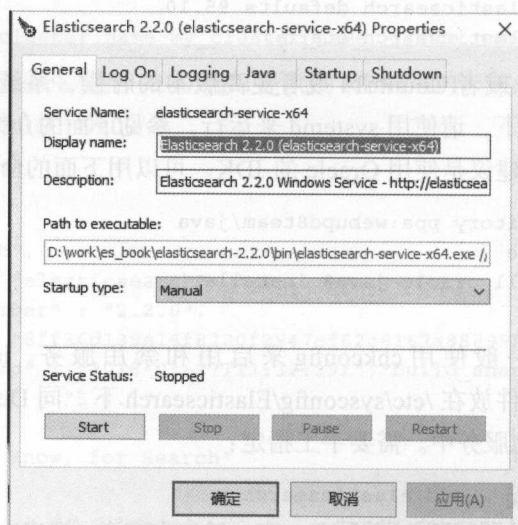


图 1-3 服务设置图形化界面

- ❑ 用命令 service start、service stop 来启动停止服务。

还有一个社区支持的可定制 MSI 安装程序：<https://github.com/salyh/elasticsearch-msi-installer> 也可以安装成服务。

1.4.7 版本升级

Elasticsearch 通常可以使用滚动升级过程，导致服务不中断。下面详细介绍如何执行滚动升级与集群的升级重启。Elasticsearch 不是所有版本都可以直接升级。升级前请先查阅相关文档，然后进行数据备份，最好在测试环境下模拟一遍。升级可以参照以下内容：

- ❑ 从 0.90.x 到 2.x 需要全集群重启。
- ❑ 从 1.x 到 2.x 需要全集群重启。
- ❑ 从 2.x 到 2.y 可以不用全部重启，使用滚动升级完成 ($y > x$)。

插件问题，在升级的时候，同时要考虑到插件问题，最好插件的版本和 Elasticsearch 版本一致。

在每个节点上升级和重启的过程为：

- 1) 关闭 Elasticsearch。
- 2) 升级 Elasticsearch。
- 3) 升级插件。
- 4) 启动 Elasticsearch。

下面介绍集群重启和滚动升级。

1. 集群重启

1) 关闭分片分配。当我们试图关闭一个节点的时候，Elasticsearch 会立即试图复制这个节点的数据到集群中的其他节点上。这将导致大量的 IO 请求。在关闭该节点的时候可以通过设置一下参数来避免此问题的发生：

```
PUT /_cluster/settings
{
  "transient": {"cluster.routing.allocation.enable": "none"}
}
```

2) 执行一个同步刷新。当停止一个索引的时候，分片的恢复会很快，所以要进行同步刷新请求。

```
POST /_flush/synced
```

同步刷新请求是非常有效的一种操作，当任何索引操作失败的时候，可以执行同步刷新请求，必要的时候可以执行多次。

3) 关闭和升级所有节点。停止在集群中的所有节点上的服务。每一个节点都要进行单独升级。这个主要就是文件替换操作，注意保留日志目录。

4) 启动集群。如果你有专门的主节点 (node.master 节点设置为 true, node.data 节点设置为 false)，则先启动主节点。等待它们形成一个集群，然后选择一个主数据节点进行启动。你可以通过查看日志来检查启动情况。通过下面命令可以监控集群的启动情况，检查所有节点是否已成功加入集群。

```
GET _cat/health
GET _cat/nodes
```

5) 等待黄色集群状态。当节点加入集群后, 它首先恢复存储在本地的主分片数据。最初的时候, 通过 `_cat/health` 请求发现集群的状态是红色, 意味着不是所有的主分片都已分配。当每个节点都恢复完成后, 集群的状态将会变成黄色, 这意味着所有主分片已经被找到, 但是并不是所有的副本分片都恢复。

6) 重新分配。延迟副本的分配直到所有节点都加入集群, 在集群的所有节点, 可以重新启用碎片分配:

```
PUT /_cluster/settings
{
  "persistent": {"cluster.routing.allocation.enable": "all"}
}
```

这个时候集群将开始复制所有副本到数据节点上, 这样可以安全地恢复索引和搜索。如果你能延迟索引和搜索直到所有的分片已经恢复, 这样可以加快集群的恢复。可以通过下面 API 监控恢复的进度和健康情况:

```
GET _cat/health
GET _cat/recovery
```

最后当集群的状态出现绿色的时候, 表示本次集群升级全部完成。

2. 滚动升级

滚动升级允许 Elasticsearch 集群升级一个节点, 同时又不影响系统的使用。在同一个集群中的所有节点的版本最好保持一致, 否则可能会产生不可预知的后果。滚动升级的步骤如下:

1) 关闭分片分配。当我们视图关闭一个节点的时候, Elasticsearch 会立即试图复制这个节点的数据到集群中的其他节点上。这将导致大量的 IO 请求。在关闭该节点的时候可以通过设置一下参数来避免此问题的发生。

```
PUT /_cluster/settings
{
  "transient": {"cluster.routing.allocation.enable": "none"}
}
```

2) 停止不必要的索引和执行同步刷新(可选)。你可以在升级过程中继续索引。如果, 暂时停止不必要的索引碎片, 但它恢复要快得多。所以可以执行同步刷新操作:

```
POST /_flush/synced
```

同步刷新请求是非常有效的一种操作, 当任何索引操作失败的时候, 可以执行同步刷新请求, 必要的时候可以执行多次。

3) 停止和升级一个节点。在启动升级前, 将节点中的一个节点关闭。可以通过绿色解压安装或者通过 RPM 等安装包安装。不管是解压安装还是压缩包安装都要保留之前的

数据文件不能被破坏。可以在新的目录中安装，把 path.conf 和 path.data 的位置指向之前的数据。

4) 启动升级节点。启动“升级”节点，并通过接口检查是否正确：

```
GET _cat/nodes
```


5) 启用共享配置。一旦节点加入集群，在节点重新启用碎片分配：

```
PUT /_cluster/settings
{
  "transient": {"cluster.routing.allocation.enable": "all"}
}
```

6) 等待节点恢复。应该在集群下一个节点升级之前完成碎片分配。可以通过以下接口进行检查：

```
GET _cat/health
```

等待的状态栏从黄到绿色。状态绿色意味着所有主分片和副本分片已经完成分配。

 **注意** 在滚动升级期间，主分片被分配到一个更高版本节点不会有副本分配到一个较低版本的节点，因为旧版本可能不能识别新版本的数据结构。

一旦另一个节点升级，副本将被分配，集群的健康状态将达到绿色。

没有同步刷新碎片可能需要一些时间来恢复。恢复的状态和每个节点的监控可以用以下接口：

```
GET _cat/recovery
```

如果你停止了索引，那么恢复索引的恢复是安全的。

7) 重复其他节点。当集群是稳定的，节点已经恢复，重复上述步骤，把所有剩余的节点进行升级。

1.5 对外接口

Elasticsearch 对外提供的 API 是以 HTTP 协议的方式，通过 JSON 格式以 REST 约定对外提供。

HTTP 配置文件是放在 elasticsearch.yml 中，注意，所有与 HTTP 配置相关的内容都是静态配置，也就是需要重启后才生效。HTTP 对外接口模块是可以禁用的，只需要设置 http.enabled 为 false 即可。Elasticsearch 集群中的通信是通过内部接口实现的，而不是 HTTP 协议。在集群中不需要所有节点都开启 HTTP 协议，正常情况下，只需要在一个节点上开启 HTTP 协议。

1.5.1 API 约定

1. 多索引参数

大多数 API 支持多索引查询，就是同时可以查询多个索引中的数据，例如，参数 test1、test2、test3 表示同时搜索 test1、test2、test3 三个索引中的数据（或者用 _all 全部索引，_all 是内部定义的关键字）。在参数中同时支持通配符的操作，例如 test* 表示查询所有以 test 开头的索引。也支持排除操作，例如 +test*，-test3 表示查询所有 test 开头的索引，排除 test3。多索引查询还支持以下参数：

- ❑ ignore_unavailable：当索引不存在或者关闭的时候，是否忽略这些索引，值为 true 和 false。
- ❑ allow_no_indices：当使用通配符查询所有索引的时候，当有索引不存在的时候是否返回查询失败。值为 true 和 false。
- ❑ expand_wildcards：控制通配符索引表达式匹配具体哪一类的索引，值为 open，close，none，all。open 表示只支持开启状态的索引，close 表示只支持关闭状态的索引，none 表示不可用，all 表示同时支持 open 和 close 索引。



注意 文档操作 API 和索引别名 API 不支持多索引参数。

2. 日期筛选

日期筛选可以限定时间序列索引的搜索范围，而不是搜索全部内容，通过时间限定，可以从集群中减少搜索的内容，提高搜索效率和减少资源占用。例如只搜索最近两天的错误日志。



注意 几乎所有的 API 都支持日期筛选。

日期筛选的语法为：

```
<static_name{date_math_expr{date_format|time_zone}}>
```

语法解释：

- ❑ static_name：索引的名称；
- ❑ date_math_expr：动态日期计算表达式；
- ❑ date_format：日期格式；
- ❑ time_zone：时区，默认为 UTC。

例如：

```
curl -XGET '127.0.0.1:9200/<logstash-{now%2Fd-2d}>/_search' {
```

```
"query" : {...}
}
```



由于 URL 编码的问题，上面的 / 被替换成了 %2F：now%2Fd-2d 被转换为 now/d-2d。

假设当前时间为 2013.7.17 日中午，下面列举几个例子：

```
<secilog-{now/d}>:secilog-2013.07.17
<secilog-{now/M}>secilog-2013.07.01
<secilog-{now/M{YYYY.MM}}>secilog-2013.07
<secilog-{now/M-1M{YYYY.MM}}>secilog-2013.06
<secilog-{now/d{YYYY.MM.dd|+12:00}}secilog-2013.7.18
```



如果索引名称有 {}，可以通过添加 \ 来转义，如：<elastic\\{ON\\}-{now/M}> 被转换为 elastic{ON}-2013.07.01

时间搜索也可以通过逗号，来选择多个时间，例如，选择最近三天的数据：

```
curl -XGET '127.0.0.1:9200/<secilog-{now%2Fd-2d}>,<secilog-{now%2Fd-1d}>,<secilog-{now%2Fd}>/_search' {
  "query" : {...}
}
```

3. 通用参数

通用参数如下：

- ❑ pretty 参数，当你在任何请求中添加了参数 ?pretty=true 时，请求的返回值是经过格式化后的 JSON 数据，这样阅读起来更加的方便。系统还提供了另一种格式的格式化，?format=yaml，YAML 格式，这将导致返回的结果具有可读的 YAML 格式。
 - ❑ human 参数，对于统计数据，系统支持计算机数据，同时也支持比较适合人类阅读的数据。比如，计算机数据 "exists_time_in_millis": 3600000 or "size_in_bytes": 1024。更适合人类阅读的数据："exists_time": "1h" or "size": "1kb"。当 ?human=false 的时候，只输出计算机数据，当 ?human=true 的时候输出更适合人类阅读的数据，但这会消耗更多的资源，默认是 false。
 - ❑ 日期表达式，大多数参数接受格式化日期表达式，如范围查询 gt(大于) 和 lt(小于)，或在日期聚合中用 from to 来表达时间范围。表达式设定的日期为 now 或者日期字符串加 ||。
 - +1h 增加一小时。
 - -1D 减少一个小时。
 - /D 上一个小时。
- 支持的时间单位为：y(年)、M(月)、w(周)、d(日)、h(小时)、m(分钟)、s(秒)。

例如:

- now+1h: 当前时间加一小时, 以毫秒为单位。
- now+1h+1m: 当前时间加一小时和一分钟, 以毫秒为单位。
- now+1h/d: 当前时间加一小时, 四舍五入到最近的一天。
- 2015-01-01||+1M/d: 2015-01-01 加一个月, 向下舍入到最近的一天。

□ 响应过滤 (filter_path)。所有的返回值可以通过 filter_path 来减少返回值的内容, 多个值可以用逗号分开。例如:

```
curl -XGET '127.0.0.1:9200/_search?pretty&filter_path=took,hits.hits._id,hits._score'{
  "took" : 3,
  "hits" : {
    "hits" : [
      { "_id" : "3640", "_score" : 1.0 },
      { "_id" : "3642", "_score" : 1.0 }
    ]
  }
}
```

它也支持通配符 * 匹配任何部分字段的名称, 例如:

```
curl -XGET '127.0.0.1:9200/_nodes/stats?filter_path=nodes.*.ho*'{
  "nodes" : {
    "lvJHed8uQQu4brS-SXKsNA" : { "host" : "portable" }
  }
}
```

我们可以用两个通配符 ** 来匹配不确定名称的字段, 例如我们可以返回 Lucene 版本的段信息:

```
curl '127.0.0.1:9200/_segments?pretty&filter_path=indices.**.version'{
  "indices" : {
    "movies" : {
      "shards" : {
        "0" : [ {
          "segments" : { "_0" : { "version" : "5.2.0" } }
        } ],
        "2" : [ {
          "segments" : { "_0" : { "version" : "5.2.0" } }
        } ]
      }
    },
    "books" : {
      "shards" : {
        "0" : [ {
          "segments" : { "_0" : { "version" : "5.2.0" } }
        } ]
      }
    }
  }
}
```



```

    }
  }
}

```

注意，有时直接返回 Elasticsearch 的某个字段的原始值，如 `_source` 字段。如果你想过滤 `_source` 字段，可以结合 `_source` 字段和 `filter_path` 参数，例如：

```

curl -XGET '127.0.0.1:9200/_search?pretty&filter_path=hits.hits._source&source=title'{
  "hits" : {
    "hits" : [ { "_source":{"title":"Book #2"}},
      { "_source":{"title":"Book #1"}},
      { "_source":{"title":"Book #3"}} ]
  }
}

```

紧凑参数 `flat_settings`，`flat_settings` 为 `true` 时候返回的内容更加紧凑，`false` 是返回的值更加的容易阅读。例如为 `true` 的时候：

```

{
  "persistent" : { },
  "transient" : {"discovery.zen.minimum_master_nodes": "1"}
}

```

为 `false` 的时候，默认的情况下为 `false`：

```

{
  "persistent" : { },
  "transient" : {
    "discovery" : {
      "zen" : {"minimum_master_nodes" : "1"}
    }
  }
}

```

4. 基于 URL 的访问控制

当多用户通过 URL 访问 Elasticsearch 索引的时候，为了防止用户误删除等操作，可以通过基于 URL 的访问控制来限制用户对某个具体索引的访问。在配置文件中添加参数：`rest.action.multi.allow_explicit_index: false`，这个参数默认为 `true`。当该参数为 `false` 时，在请求参数中指定具体索引的请求将会被拒绝。

1.5.2 REST 介绍

REST (REpresentational State Transfer) 从字面看就是“表述性状态传输”，它通常是开发的一种约定，当所有的开发者都遵从这种约定的时候，可以大大简化开发的沟通成本。REST 约定用 HTTP 的请求头 POST、GET、PUT、DELETE 正好可以对应 CRUD (Create、Read、Update、Delete) 四种数据操作。如果你设计的应用程序能符合 REST 原则 (REST

可称为 "RESTful Web Service" 也称 "RESTful Web API"。REST 请求头说明见表 1-3。

表 1-3 REST 请求头

HTTP 方法	数据处理	说明
POST	Create	新增一个没有 ID 的资源
GET	Read	取得一个资源
PUT	Update	更新一个资源。或新增一个含 ID 的资源（如果 ID 不存在）
DELETE	Delete	删除一个资源

1.5.3 Head 插件安装

工欲善其事必先利其器，在学习 Elasticsearch 的过程中也要借助一些工具来进行。官方文档中的模拟工具用的是 curl，这是控制台工具，不是很直观。所以在本书中用到了 Head 插件来作为请求的工具。

插件安装，在 elasticsearch/bin 目录下执行下面的命令：

```
plugin install mobz/elasticsearch-head
```

安装成功后，启动 Elasticsearch，然后在浏览器中输入：

```
http://127.0.0.1:9200/_plugin/head/
```

可以看到如图 1-4 所示的界面，表示安装成功。

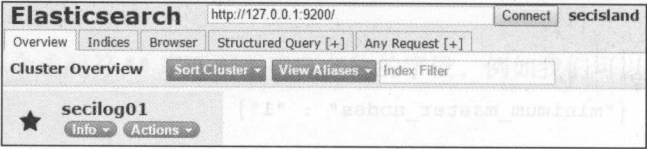


图 1-4 Head 插件整体视图

在本书大多数的接口操作中使用 Any Request 面板进行交互，界面参见图 1-5。

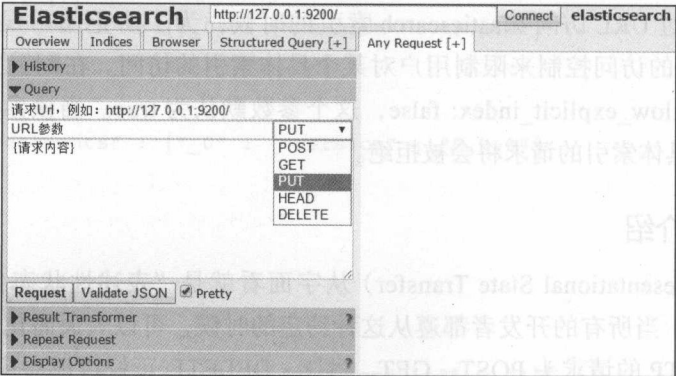


图 1-5 Head 插件请求视图



注意 在请求 URL 中要用 127.0.0.1 或者绑定的具体 IP 地址，用 localhost 不起作用。

1.5.4 创建库

在请求 URL 中输入：`http://127.0.0.1:9200/secisland?pretty`。

在请求的方法中选择 PUT，出现图 1-6 所示的界面。

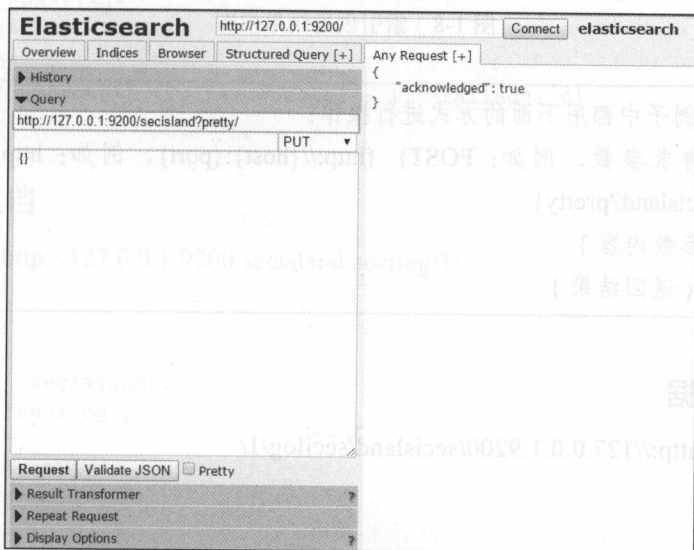


图 1-6 创建索引库示意图

点击 Request 按钮后可以在右边看到返回的内容如下，表示建库成功：

```
{
  "acknowledged" : true
}
```

执行完建库后查询一下库的状态，有两种方式查看，如果用命令查看，可以在浏览器中执行 `http://127.0.0.1:9200/_cat/indices?v`，返回结果如图 1-7 所示。

health	status	index	pri	rep	docs.count	docs.deleted	store.size	pri.store.size
yellow	open	secisland	5	1	0	0	650b	650b

图 1-7 索引库状态查询结果

表示已经建成了一个索引 secisland，主分片是 5 个，健康度是黄色，状态是活动，文档数为 0。

或者在 Head 插件中进行查看，如图 1-8 所示。

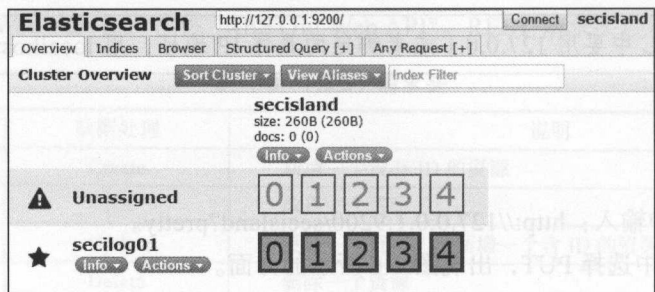


图 1-8 索引创建结果视图

注意 在后面的例子中都用下面的方式进行操作：

请求：{ 请求参数，例如：POST } {http://{host}:{port}，例如：http://127.0.0.1:9200/secisland?pretty}

参数：{ 参数内容 }

返回值：{ 返回结果 }

1.5.5 插入数据

请求：PUT http://127.0.0.1:9200/secisland/secilog/1/

参数：

```
{
  "computer": "secisland",
  "message": "secisland is an security company!"
}
```

返回值：

```
{
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 1,
  "_shards": { "total": 2, "successful": 1, "failed": 0 },
  "created": true
}
```

1.5.6 修改文档

请求：POST http://127.0.0.1:9200/secisland/secilog/1/_update

参数：

```
{
```



```

"doc": {
  "computer": "secisland",
  "message": "secisland is an security computer. It provides log analysis products !"
}
}

```

返回值:

```

{
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 2,
  "_shards": {"total": 2, "successful": 1, "failed": 0}
}

```

1.5.7 查询文档

请求: GET http://127.0.0.1:9200/secisland/secilog/1/

返回值:

```

{
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 2,
  "found": true,
  "_source": {
    "computer": "secisland",
    "message": "secisland is an security computer. It provides log analysis
products !"
  }
}

```

1.5.8 删除文档

请求: DELETE http://127.0.0.1:9200/secisland/secilog/1/

返回值:

```

{
  "found": true,
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 3,
  "_shards": {"total": 2, "successful": 1, "failed": 0}
}

```

1.5.9 删除库

请求: DELETE http://127.0.0.1:9200/secisland/

返回值:

```
{"acknowledged": true}
```

1.6 Java 接口

Elasticsearch 本身是 Java 开发的,天生对 Java 的支持能力是最好的,所以用 Java 来开发 Elasticsearch 应该是一个不错的选择。

1.6.1 Java 接口说明

可以把 Java 开发看成 CS 模式,即客户端请求,服务端响应,所有的操作是完全异步的。此外在客户端上操作,很多的操作可以在客户端上来完成,增加了系统效率。Java 客户端和服务端都是使用相同的程序。在 5.0.0 版本中,增加了单独的 HTTP REST 客户端。

1. mvn 仓库配置

在 mvn 项目中可以在 pom.xml 增加 mvn 仓库,如下所示。

```
<dependency>
  <groupId>org.elasticsearch</groupId>
  <artifactId>elasticsearch</artifactId>
  <version>${es.version}</version>
</dependency>
```

打成独立的 jar 包。如果你想打成独立的 jar 包,并包含所有的依赖。不能使用 mvn 的 maven-assembly-plugin 插件,因为它不能很好地处理 META-INF/services 结构。我们可以用 maven-shade-plugin 插件来进行打包。例如:

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>2.4.1</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals><goal>shade</goal></goals>
      <configuration>
        <transformers>
          <transformer implementation="org.apache.maven.plugins.shade.
            resource.ServicesResourceTransformer"/>
        </transformers>
      </configuration>
    </execution>
  </executions>
```

```
</executions></plugin>
```

如果你有一个 main 函数，可以通过调用 `java -jar yourjar.jar` 运行时，只需要在插件配置中加入一句话。例如：

```
<transformer implementation="org.apache.maven.plugins.shade.resource.
ManifestResourceTransformer">
  <mainClass>com.secisland.es.demo.main</mainClass>
</transformer>
```

可以使用多种方式来调用 Java 客户端。一种方式是把节点加入集群，不存储数据。另一种是把节点作为客户端连接到集群。



警告 客户端版本要和集群版本一致，否则有可能出现不可预知的错误。

2. 嵌入式节点客户端

实例化一个基于节点的客户，这是对 Elasticsearch 执行操作的最简单的方法。例如：

```
import static org.elasticsearch.node.NodeBuilder.*;
// on startup
Node node = nodeBuilder().node();
Client client = node.client();
// on shutdown
node.close();
```

当你开始一个节点，它加入了一个 Elasticsearch 集群。你可以通过设置 `cluster.name` 来配置不同的集群，或显式使用 `clusterName` 方法来创建对象。在项目工程中可以定义 `cluster.name`，配置在 `/src/main/resources/elasticsearch.yml` 文件中。只要在 classpath 中能找到 `elasticsearch.yml` 文件，节点启动的时候就会加载此文件。配置文件中定义：

```
cluster.name: secilog
```

或者在 Java 代码中指定：

```
Node node = nodeBuilder().clusterName("secilog").node();
Client client = node.client();
```

使用客户端的好处是依赖 Elasticsearch 服务端的特性把需要执行的操作自动路由到节点，当加入集群节点作为客户端时，最重要的是不能在此节点上增加数据保存，这可以通过配置来完成。把 `node.data` 设置为 `false`，或将 `node.client` 设置为 `true`。这都可以通过 `NodeBuilder` 的方法来实现。

```
import static org.elasticsearch.node.NodeBuilder.*;
```

```
// on startup
// 嵌入式节点可以不用打开 http 端口
Node node =
```

```

nodeBuilder()
    .settings(Settings.settingsBuilder().put("http.enabled", false))
    .client(true)
    .node();
Client client = node.client();
// on shutdown
node.close();

```

另一个常见的客户端用处是用在单元/集成测试中。在这种情况下，只需要启动一个“本地”（local）节点。这里的 local 节点的意思是公用一个 Java 虚拟机。在这种情况下两个服务会互相发现自己并形成集群。例如：

```

import static org.elasticsearch.node.NodeBuilder.*;
// on startup
Node node = nodeBuilder().local(true).node();
Client client = node.client();
// on shutdown
node.close();

```

嵌入式节点的缺点是：

- ❑ 频繁启动和停止一个或多个节点在集群上会创建不必要的噪声。
- ❑ 就像其他节点一样嵌入式节点客户端会响应外部请求。



注意 用 Eclipse 开发的时候有可能会报错：

```
Exception in thread "main" java.lang.IllegalStateException: path.home is not
configured
```

解决方法：在 main 方法的类中选择 run configurations，在 Arguments 标签页下的 vm arguments 输入：-Des.path.home=-Des.path.home 即可。例如：

```
-Des.path.home=D:\elasticsearch-2.2.0
```

3. 通过连接的客户端

客户端通过 TransportClient 对象可以使用远程连接的方式连接 Elasticsearch 集群。这种情况下不用加入集群，比较像传统的 CS 程序的架构，如数据库连接。例如：

```

// on startup
Client client = TransportClient.builder().build().addTransportAddress(new Inet
SocketTransportAddress(InetAddress.getByName("host1"), 9300))
    .addTransportAddress(new InetSocketTransportAddress(InetAddress.
getByName("host2"), 9300));
// on shutdown
client.close();

```

注意，如果集群名称不是“Elasticsearch”，你必须设置集群名称或者使用 elasticsearch.yml 配置文件在客户端工程中配置。可以连接一个集群中的多个节点。


```
Settings settings = Settings.settingsBuilder()
    .put("cluster.name", "secilog").build();
Client client = TransportClient.builder().settings(settings).build();
```

可以设置 `client.transport.sniff` 为 `true` 来使客户端去嗅探整个集群的状态，把集群中其他机器的 IP 地址加到客户端中。这样做的好处是一般你不用手动设置集群里所有集群的 IP 到连接客户端，它会自动帮你添加，并且自动发现新加入集群的机器。

```
Settings settings = Settings.settingsBuilder()
    .put("client.transport.sniff", true).build();
TransportClient client = TransportClient.builder().settings(settings).build();
```

其他参数说明如下：

- ❑ `client.transport.ignore_cluster_name`：设置为 `true` 的时候忽略连接节点时的集群名称验证。
- ❑ `client.transport.ping_timeout`：等待一个节点的 ping 响应的的时间，默认 5 秒。
- ❑ `client.transport.nodes_sampler_interval`：监听和连接节点的频率（时间间隔），默认 5 秒。

1.6.2 创建索引文档

引入的头文件：

```
import static org.elasticsearch.node.NodeBuilder.nodeBuilder;

import java.io.IOException;
import java.net.InetAddress;
import java.util.Date;
import java.util.Map;
import java.util.Set;

import org.elasticsearch.action.admin.cluster.health.ClusterHealthResponse;
import org.elasticsearch.action.admin.indices.create.CreateIndexRequestBuilder;
import org.elasticsearch.action.admin.indices.create.CreateIndexResponse;
import org.elasticsearch.action.index.IndexResponse;
import org.elasticsearch.client.Client;
import org.elasticsearch.client.ClusterAdminClient;
import org.elasticsearch.client.transport.TransportClient;
import org.elasticsearch.cluster.health.ClusterIndexHealth;
import org.elasticsearch.common.settings.Settings;
import org.elasticsearch.common.transport.InetSocketTransportAddress;
import org.elasticsearch.common.xcontent.XContentBuilder;
import org.elasticsearch.common.xcontent.XContentFactory;
import org.elasticsearch.node.Node;
import static org.elasticsearch.common.xcontent.XContentFactory.*;

// 上面的内容对下面的几个示例同样起作用
```

```

XContentBuilder mapping = XContentFactory.jsonBuilder()
    .startObject()
    .startObject("settings")
    .field("number_of_shards", 1) // 设置分片数量
    .field("number_of_replicas", 0) // 设置副本数量
    .endObject()
    .endObject()
    .startObject()
    .startObject(type) // type 名称
    .startObject("properties") // 下面是设置文档列属性
    .startObject("type").field("type", "string").field("store", "yes")
    .endObject()
    .startObject("eventCount").field("type", "long").field("store", "yes")
    .endObject()
    .startObject("eventDate").field("type", "date")
    .field("format", "dateOptionalTime").field("store", "yes")
    .endObject()
    .startObject("message").field("type", "string")
    .field("index", "not_analyzed").field("store", "yes")
    .endObject()
    .endObject()
    .endObject()
    .endObject();

```

```

CreateIndexRequestBuilder cirb = client.admin().indices()
    .prepareCreate(indexName) // index 名称
    .setSource(mapping);

```

```

CreateIndexResponse response = cirb.execute().actionGet();
if (response.isAcknowledged()) {
    System.out.println("Index created.");
} else {System.err.println("Index creation failed.");}

```

1.6.3 增加文档

代码如下：

```

IndexResponse response = client
    .prepareIndex(indexName, type, "1")
    .setSource(// 这里可以直接用 json 字符串
        jsonBuilder().startObject()
            .field("type", "syslog")
            .field("eventCount", 1)
            .field("eventDate", new Date())
            .field("message", "secilog insert doc test")
            .endObject()).get();
System.out.println("index:"+response.getIndex()
    + " insert doc id:"+response.getId())

```

```
+" result:"+response.isCreated());
```

1.6.4 修改文档

修改文档有两种方式，一种是直接修改，另一种是如果文档不存在则插入存在再修改。

第一种方式的代码：

```
UpdateRequest updateRequest = new UpdateRequest();
updateRequest.index(indexName);
updateRequest.type(type);
updateRequest.id("1");
updateRequest.doc(jsonBuilder()
    .startObject().field("type", "file").endObject());
client.update(updateRequest).get();
```

第二种方式的代码：

```
IndexRequest indexRequest = new IndexRequest(indexName, type, "3")
    .source(jsonBuilder()
        .startObject()
            .field("type", "syslog")
            .field("eventCount", 2)
            .field("eventDate", new Date())
            .field("message", "secilog insert doc test")
        .endObject());
UpdateRequest updateRequest = new UpdateRequest(indexName, type, "3")
    .doc(jsonBuilder().startObject().field("type", "file").endObject());
client.update(updateRequest).get();
```

1.6.5 查询文档

代码如下：

```
GetResponse response = client.prepareGet("secilog", "log", "1").get();
String source = response.getSource().toString();
long version = response.getVersion();
String indexName = response.getIndex();
String type = response.getType();
String id = response.getId();
```

1.6.6 删除文档

删除文档如下：

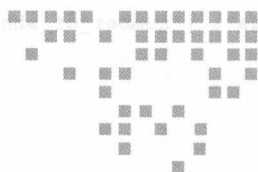
```
DeleteResponse dresponse = client.prepareDelete("secilog", "log", "4").get();
boolean isFound = dresponse.isFound(); // 文档存在返回 true, 不存在返回 false
```

删除索引如下：

```
DeleteIndexRequest delete = new DeleteIndexRequest("secilog");
client.admin().indices().delete(delete);
```

1.7 小结

本章介绍了全文搜索 Elasticsearch 的基本概念和历史,介绍了需要掌握 Elasticsearch 的基础知识。Elasticsearch 的安装、配置和升级操作,使用 Elasticsearch REST API 和 Java 接口来索引、更新、搜索、删除数据。通过学习本章,应该对 Elasticsearch 有了基本的认识。



第2章

Chapter 2

索引

索引是具有相同结构的文档集合。在 Elasticsearch 中索引是个非常重要的内容，对 Elasticsearch 的大部分操作都是基于索引来完成的。本章介绍索引的映射、设置、监控管理，以及索引的状态和文档操作管理。

2.1 索引管理

第 1 章里已经启动了集群并创建了默认的索引，除了默认的索引配置外，我们还可以通过索引管理来对索引进行更高级别的操作。首先了解一下创建索引的过程，后面将介绍其他相关操作。

2.1.1 创建索引

创建索引的时候可以通过修改 `number_of_shards` 和 `number_of_replicas` 参数的数量来修改分片和副本的数量。在默认的情况下分片的数量是 5 个，副本的数量是 1 个。

例如，创建三个主分片，两个副本分片的索引。

请求：PUT <http://127.0.0.1:9200/secisland/>

参数：

```
{
  "settings" : {
    "index" : {"number_of_shards" : 3, "number_of_replicas" : 2}
  }
}
```

参数可以简写成：

```
{
  "settings": { "number_of_shards": 3, "number_of_replicas": 2 }
}
```

返回值:

```
{
  "acknowledged": true
}
```

然后我们直观地看一下创建后的效果,如图 2-1 所示。

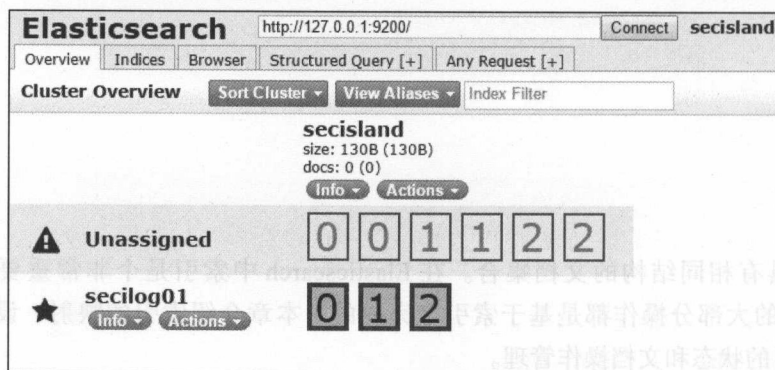


图 2-1 修改分片数量示意图

后面可以通过 update-index-settings API 完成对副本数量的修改。例如

请求: PUT http://127.0.0.1:9200/secisland/_settings/

参数:

```
{
  "number_of_replicas": 1
}
```

这样就把副本的数量改成 1 个副本。

对于任何 Elasticsearch 文档而言,一个文档会包括一个或者多个字段,任何字段都要有自己的数据类型,例如 string、integer、date 等。Elasticsearch 中是通过映射来进行字段和数据类型对应的。在默认的情况下 Elasticsearch 会自动识别字段的数据类型。同时 Elasticsearch 提供了 mappings 参数可以显式地进行映射。

我们看一下在 Lucene 中是如何看待文档的。在 Lucene 中的文档包含的是一个简单 field-value 对的列表。一个字段至少要有个值,但是任何字段都可以拥有多个值。类似地,一个字符串值也可以通过解析阶段而被转换为多个值。Lucene 不管值是字符串类型,还是数值类型或者其他类型,所有的值都会被同等地看做一些不透明的字节 (Opaque bytes)。当我们使用 Lucene 对文档进行索引时,每个字段的值都会被添加到倒排索引 (Inverted Index) 的对应字段中。也可以选择原始值是否会不修改地被保存到索引中,以此

来方便将来获取。

创建自定义字段类型的例子如下：

请求：PUT <http://127.0.0.1:9200/secisland>

参数：

```
{
  "settings": {"number_of_shards": 3, "number_of_replicas": 2},
  "mappings": {"secilog": {"properties": {"logType": {"type": "string",
    "index": "not_analyzed"}}}}}
}
```

在这个例子中，我们创建了一个名为 secilog 的类型，类型中有一个字段，字段的名称是 logType，字段的数据类型是 string，而且这个字段是不进行分析的。

2.1.2 删除索引

请求：DELETE <http://127.0.0.1:9200/secisland/>

上面的示例删除了名为 secisland 的索引。删除索引需要指定索引名称，别名或者通配符。

删除索引可以使用逗号分隔符，或者使用 _all 或 * 号删除全部索引。



注意 _all 或 * 删除全部索引时要谨慎操作。

为了防止误删除，可以设置 elasticsearch.yml 属性 action.destructive_requires_name 为 true，禁止使用通配符或 _all 删除索引，必须使用名称或别名才能删除该索引。

2.1.3 获取索引

获取索引接口允许从一个或多个索引中获取信息。

请求：GET <http://127.0.0.1:9200/secisland/>

通过这个请求会把系统中的信息都显示出来，包括默认的一些配置，例如上面的返回值为：

```
{
  "secisland": {
    "aliases": { },
    "mappings": {
      "secilog": {
        "properties": {
          "logType": {"type": "string", "index": "not_analyzed"}
        }
      }
    }
  },
}
```

```

"settings": {
  "index": {
    "creation_date": "1459652280258",
    "uuid": "2EsTzGu-QuCh7ddPGZ9gUA",
    "number_of_replicas": "2",
    "number_of_shards": "3",
    "version": {"created": "2020099"}
  },
  "warmers": { }
}

```

上面的示例获取名为 `secisland` 的索引。获取索引需要指定索引名称，别名或者通配符。获取索引可以使用通配符获取多个索引，或者使用 `_all` 或 `*` 号获取全部索引。

返回结果过滤：可以自定义返回结果的属性。

请求：GET http://127.0.0.1:9200/secisland/_settings,_mappings

上面示例只返回 `secisland` 索引的 `settings` 和 `mappings` 属性。可配置的属性包括 `_settings`、`_mappings`、`_warmers` 和 `_aliases`。

如果索引不存在，系统会返回一个错误内容，例如：

请求：GET <http://127.0.0.1:9200/secilog>

```

{
  "error": {
    "root_cause": [
      {
        "type": "index_not_found_exception",
        "reason": "no such index",
        "index": "secilog",
        "resource.type": "index_or_alias",
        "resource.id": "secilog"
      }
    ],
    "type": "index_not_found_exception",
    "reason": "no such index",
    "index": "secilog",
    "resource.type": "index_or_alias",
    "resource.id": "secilog"
  },
  "status": 404
}

```

2.1.4 打开/关闭索引

打开/关闭索引接口允许关闭一个打开的索引或者打开一个已经关闭的索引。关闭的索引只能显示索引元数据信息，不能够进行读写操作。

打开/关闭索引的方式是 `{/索引名}/_close` 或者 `{/索引名}/_open`，完整示例如下。

请求: POST 127.0.0.1:9200/secisland/_close, 关闭索引示意图, 如图 2-2 所示。

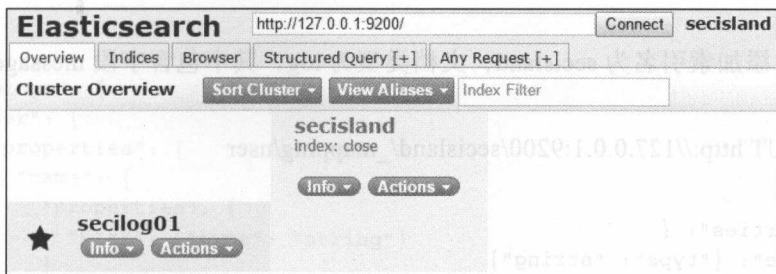


图 2-2 关闭索引示意图

请求: POST 127.0.0.1:9200/secisland/_open

可以同时打开或关闭多个索引。如果指向不存在的索引会抛出错误。可以使用配置 `ignore_unavailable=true`, 不显示异常。

全部索引可以使用 `_all` 打开或关闭, 或者使用通配符表示全部 (比如 `*`)

设置 `config/elasticsearch.yml` 属性 `action.destructive_requires_name` 为 `true`, 禁止使用通配符或者 `_all` 标识索引。

因为关闭的索引会继续占用磁盘空间而不能使用, 所以关闭索引接口可能造成磁盘空间的浪费。

禁止使用关闭索引功能, 可以设置 `settingscluster.indices.close.enable` 为 `false`, 默认是 `true`。

2.2 索引映射管理

在前面例子中 Elasticsearch 创建文档的时候, 是没有指定索引参数, 这个时候系统会自动判断每个维度的类型, 有很多时候需要我们进行一些更高级的设置, 比如索引分词, 是否存储等。

2.2.1 增加映射

API 允许你向索引 (index) 添加文档类型 (type), 或者向文档类型 (type) 中添加字段 (field)。

请求: PUT http://127.0.0.1:9200/secisland

参数:

```
{
  "mappings": {
    "log": {
      "properties": {
        "message": {"type": "string"}
      }
    }
  }
}
```

```

    }
  }
}

```

以上接口添加索引名为 secisland，文档类型为 log，其中包含字段 message，字段类型是字符串：

请求：PUT http://127.0.0.1:9200/secisland/_mapping/user

```

{
  "properties": {
    "name": {"type": "string"}
  }
}

```

向已经存在的索引 secisland 添加文档类型为 user，包含字段 name，字段类型是字符串。

请求：PUT http://127.0.0.1:9200/secisland/_mapping/log

```

{
  "properties": {
    "user_name": {"type": "string"}
  }
}

```

已经存在的索引 secisland，文档类型为 log，添加新的字段 user_name，字段类型是字符串。

1. 多个索引设置映射

可以一次向多个索引添加文档类型。

请求：PUT http://127.0.0.1:9200/{index}/_mapping/{type}

{body}

{index} 可以有多种方式，逗号分隔：比如 test1,test2,test3。_all 表示所有索引。通配符 * 表示所有。test* 表示以 test 开头。

{type} 需要添加或更新的文档类型。

{body} 需要添加的字段或字段类型。

2. 更新字段映射

在一般情况下，对现有字段的映射不会更新。对这个规则有一些例外。例如：

❑ 新的属性被添加到对象数据类型的字段。

❑ 新的多域字段被添加到现有的字段。

❑ doc_values 可以被禁用。

❑ 增加了 ignore_above 参数。

例如：

请求: PUT http://127.0.0.1:9200/secisland

参数:

```
{
  "mappings": {
    "user": {
      "properties": {
        "name": {
          "properties": {
            "first": {"type": "string"}
          }
        },
        "user_id": {"type": "string", "index": "not_analyzed"}
      }
    }
  }
}
```

user 的第一个 name 属性是对象数据类型 (Object datatype) 字段, 对上个索引进行修改。

请求: PUT http://127.0.0.1:9200/secisland/_mapping/user

参数:

```
{
  "properties": {
    "name": {
      "properties": {
        "last": {"type": "string"}
      }
    },
    "user_id": {"type": "string", "index": "not_analyzed", "ignore_above": 100}
  }
}
```

修改映射, 对第一个对象数据类型增加了一个属性是 last。修改了 user_id, 通过设置 ignore_above 使默认的更新为 100。

3. 不同类型之间的冲突

在同一个索引的不同类型中, 相同名称的字段中必须有相同的映射, 因为它们内部是在同一个领域内, 如果试图在这种情况下更新映射参数, 系统将会抛出异常。除非在更新的时候指定 update_all_types 参数。在这种情况下它将更新所有同一索引同名称的映射参数。

例如:

请求: PUT http://127.0.0.1:9200/secisland

参数:

```
{
  "mappings": {
```

```

    "type_one": {
      "properties": {
        "text": { "type": "string", "analyzer": "standard" }
      }
    },
    "type_two": {
      "properties": {
        "text": { "type": "string", "analyzer": "standard" }
      }
    }
  }
}

```

修改映射的操作如下。

请求: PUT http://127.0.0.1:9200/secisland/_mapping/type_one

参数:

```

{
  "properties": {
    "text": {
      "type": "string",
      "analyzer": "standard",
      "search_analyzer": "whitespace"
    }
  }
}

```

此时会抛出异常, 然后增加参数 `update_all_types`, 同时更新两个类型。

请求: PUT http://127.0.0.1:9200/secisland/_mapping/type_one?update_all_types

参数:

```

{
  "properties": {
    "text": {
      "type": "string",
      "analyzer": "standard",
      "search_analyzer": "whitespace"
    }
  }
}

```

2.2.2 获取映射

获取文档映射接口允许通过索引或者索引和类型来搜索:

GET http://127.0.0.1:9200/secisland/_mapping/secilog

系统同时支持获取多个索引和类型的语法。

获取文档映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式:

`host:port/{index}/_mapping/{type}`, `{index}` 和 `{type}` 可以接受逗号 (,) 分隔符, 也可以使用

`_all` 来表示全部索引。如下所示：

```
GET http://127.0.0.1:9200/_mapping/secisland,kimchy
GET http://127.0.0.1:9200/_all/_mapping/secilog,book
```

第一个省略 `_all`，第二个使用 `_all`，两者都是表示全部索引。也就是说，下面两个是等价的：

```
GET http://127.0.0.1:9200/_all/_mapping
GET http://127.0.0.1:9200/_mapping
```

2.2.3 获取字段映射

获取文档字段接口允许你搜索一个或多个字段。这个用来搜索想要搜索的字段，而不是某个索引或者文档类型的全部内容。

这段请求只返回字段为 `text` 的内容：

```
GET http://127.0.0.1:9200/secisland/_mapping/secilog/field/text
```

返回值如下（假定 `text` 为 `String` 类型）：

```
{
  "secisland": {
    "secilog": {
      "text": {
        "full_name": "text",
        "mapping": {"text": { "type": "string" }}
      }
    }
  }
}
```

获取多索引和类型的字段映射。

获取文档字段映射接口一次可以获取多个索引或文档映射类型。该接口通常是如下格式：

```
host:port/{index}/{type}/_mapping/field/{field}
```

`{index}`，`{type}`，`{field}` 可以使用逗号 (,) 分隔，也可以使用通配符。

其中 `{index}` 可以使用 `_all` 表示全部索引，示例如下：

```
GET http://127.0.0.1:9200/secisland,kimchy/_mapping/field/message
GET http://127.0.0.1:9200/_all/_mapping/secilog,book/field/message,user.id
GET http://127.0.0.1:9200/_all/_mapping/tw*/field/*.id
```

指定字段的操作如下。获取文档字段接口，可以使用逗号 (,) 分隔符或者通配符 (*)。

如下文档示例，如果只使用字段名 `id` 会产生歧义：

```
{
  "article": {
```

```

"properties": {
  "id": { "type": "string" },
  "title": { "type": "string" },
  "abstract": { "type": "string" },
  "author": {
    "properties": { "id": { "type": "string" }, "name": { "type": "string" } }
  }
}
}
}

```

如果想要表示 author 中的 id 和 name, 可使用 author.id 和 author.name。请求如下:

```
GET http://127.0.0.1:9200/secisland/_mapping/article/field/
author.id,abstract,author.name
```

返回值如下:

```

{
  "secisland": {
    "article": {
      "abstract": {
        "full_name": "abstract",
        "mapping": { "abstract": { "type": "string" } }
      },
      "author.id": {
        "full_name": "author.id",
        "mapping": { "id": { "type": "string" } }
      },
      "author.name": {
        "full_name": "author.name",
        "mapping": { "name": { "type": "string" } }
      }
    }
  }
}

```

2.2.4 判断类型是否存在

检查索引或文档类型是否存在:

```
HEAD http://127.0.0.1:9200/secisland/secilog
```

存在返回 200, 不存在返回 404。

2.3 索引别名

在 Elasticsearch 所有的 API 中, 对应的是一个或者多个索引。Elasticsearch 可以对一个或者多个索引指定别名, 通过别名可以查询到一个或者多个索引的内容。在内部, Elasticsearch 会自动把别名映射到相应的索引上。可以对别名编写过滤器或者路由, 在系统

中别名不能重复,也不能和索引名重复。其实 Elasticsearch 的别名机制有点像数据库中的视图。例如:为索引 test1 增加一个别名 alias1。

请求: POST http://127.0.0.1:9200/_aliases

参数:

```
{
  "actions" : [{ "add" : { "index" : "secisland", "alias" : "alias1" } }]
}
```

删除别名:与请求是一样的,参数不一样:

```
{
  "actions" : [{ "remove" : { "index" : "test1", "alias" : "alias1" } }]
}
```

注意:别名没有修改的语法,当需要修改别名的时候,可以先删除别名,然后再增加别名,例如:

```
{
  "actions" : [
    { "remove" : { "index" : "test1", "alias" : "alias1" } },
    { "add" : { "index" : "test1", "alias" : "alias2" } }
  ]
}
```

一个别名关联多个索引:

```
{
  "actions" : [
    { "add" : { "index" : "test1", "alias" : "alias1" } },
    { "add" : { "index" : "test2", "alias" : "alias1" } }
  ]
}
```

或者用下面的语法:

```
{
  "actions" : [
    { "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }
  ]
}
```

或者使用通配符:

```
{
  "actions" : [
    { "add" : { "index" : "test*", "alias" : "all_test_indices" } }
  ]
}
```



注意 通配符指定的索引只是在当前生效,后面添加的索引不会被自动添加到别名上。对某一别名做索引的时候,如果该别名关联多个索引会报错。

1. 过滤索引别名

通过过滤索引来指定别名提供了对索引查看的不同视图，该过滤器可以使用查询 DSL 来定义适用于所有的搜索，计数，查询删除等，以及更多类似这样的与此别名的操作。

要创建一个带过滤的别名，首先我们需要确保映射中已经存在需要过滤的字段：

创建一个索引，请求：PUT <http://127.0.0.1:9200/test1>。

参数：

```
{
  "mappings": {
    "type1": {
      "properties": {
        "user" : { "type": "string", "index": "not_analyzed" }
      }
    }
  }
}
```

创建过滤别名，请求：POST http://127.0.0.1:9200/_aliases。

参数：

```
{
  "actions" : [
    { "add" : { "index" : "test1", "alias" : "alias2",
               "filter" : { "term" : { "user" : "kimchy" } } } }
  ]
}
```

通过别名也可以和路由关联，此功能可以和过滤别名命令一起使用，以避免不必要的碎片操作。例如：

请求：POST http://127.0.0.1:9200/_aliases

参数：

```
{
  "actions" : [
    { "add" : { "index" : "test", "alias" : "alias1", "routing" : "1" } }
  ]
}
```

同时可以指定搜索路由或者查询路由，例如参数：

```
{
  "actions" : [
    { "add" : { "index" : "test", "alias" : "alias2", "search_routing" : "1,2",
               "index_routing" : "2" } }
  ]
}
```




注意 搜索路由可以指定多个值，索引路由只能指定一个值。如果使用路由别名操作的同时还有路由参数，则结果是别名路由和路由的交集。例如以下命令将使用“2”作为路由值：

```
GET http://127.0.0.1:9200/alias2/_search?q=user:kimchy&routing=2,3
```

通过参数添加别名。

语法：请求 PUT `/_{index}/_alias/{name}`

参数：`{"routing": "1", "filter"}`

条件解释：

- `index`：参照的索引，可以使用 `*`，`_all`，正则表达式或者逗号分开的多个 `name1`，`name2`，...
- `name`：别名的名称。
- `routing`：别名对应的路由。
- `filter`：指定别名时候的过滤条件。

例如：PUT `http://127.0.0.1:9200/secisland/_alias/2013`

例如一个索引。

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings" : {
    "secilog" : {
      "properties" : {"user_id" : {"type" : "integer"}}
    }
  }
}
```

指定别名：

请求：PUT `http://127.0.0.1:9200/secisland/_alias/user_12`

```
{
  "routing" : "12",
  "filter" : {
    "term" : {"user_id" : 12}
  }
}
```

建索引的时候同时指定别名

例如：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings" : {
```

```

"type" : {
  "properties" : {"year" : {"type" : "integer"}}
},
"aliases" : {
  "current_day" : {},
  "2014" : {
    "filter" : {"term" : {"year" : 2014 }}
  }
}
}

```

2. 删除别名

语法: DELETE http://{host}:{port} /{index}/_alias/{name}

例如: DELETE http://127.0.0.1:9200/secisland/_alias/user_12

3. 查询现有的别名

可以通过索引名或者别名进行查询。参数如下:

- ❑ index: 索引别名的名称。部分名称支持通配符, 用逗号分隔也可以指定多个索引名称, 还可以使用索引的别名名称。
- ❑ alias: 在响应中返回的别名名称。该参数支持通配符和用逗号分隔的多个别名。
- ❑ ignore_unavailable: 如果一个指定的索引名称不存在该怎么办。如果设置为 true, 那么这些索引将被忽略。

语法: GET http://{host}:{port}/{index}/_alias/{alias}

例如: GET http://127.0.0.1:9200/secisland/_alias/*

返回值:

```

{
  "secisland" : {
    "aliases" : {
      "user_13" : {
        "filter" : {"term" : {"user_id" : 13}},
        "index_routing" : "13",
        "search_routing" : "13"
      },
      "user_14" : {
        "filter" : {"term" : {"user_id" : 14}},
        "index_routing" : "14",
        "search_routing" : "14"
      }
    }
  }
}

```

下面的例子中包括所有别名为 2013 的。

请求: GET http://127.0.0.1:9200/_alias/2013

返回值:

```
{
  "logs_201304" : {
    "aliases" : { "2013" : { } }
  },
  "logs_201305" : {
    "aliases" : { "2013" : { } }
  }
}
```

请求: GET http://127.0.0.1:9200/_alias/2013_01*

返回:

```
{
  "logs_20130101" : {
    "aliases" : { "2013_01" : { } }
  }
}
```

用 HEAD 也可以检查别名是否存在,语法和 GET 类似,例如:

```
HEAD http://127.0.0.1:9200/_alias/2013
HEAD http://127.0.0.1:9200/_alias/2013_01*
HEAD http://127.0.0.1:9200/users/_alias/*
```

2.4 索引配置

在 Elasticsearch 中索引有很多的配置参数,有些配置是可以在建好索引后重新进行设置和管理的,比如索引的副本数量、索引的分词等,本节主要介绍索引的参数设置。

2.4.1 更新索引配置

在 REST 风格的 URL 设置中设置 /_settings (所有索引) 或者 {index}/_settings, 可以设置一个或者多个索引,例如:

请求: PUT http://127.0.0.1:9200/secisland/_settings

参数:

```
{
  "index" : { "number_of_replicas" : 4 }
}
```

更新分词器。创建索引后可以添加新的分析器。添加分析器之前必须先关闭索引,添加之后再打开索引。

```
POST http://127.0.0.1:9200/secisland/_close
PUT http://127.0.0.1:9200/secisland/_settings
```

参数:

```
{
  "analysis" : {
    "analyzer":{
      "content":{"type":"custom","tokenizer":"whitespace"}
    }
  }
}
```

POST http://127.0.0.1:9200/secisland/_open

如上示例, 先关闭 secisland 索引, 然后添加自定义分析器, 分析器策略是空格分析器 (whitespace), 就是按照空格进行分词。

2.4.2 获取配置

索引中包含很多配置参数, 可以通过下面命令获取索引的参数配置:

```
GET http://127.0.0.1:9200/secisland/_settings
```

获取索引配置参数的请求格式如下:

```
host:port/{index}/_settings
```

{index} 为索引名称, 可以接收多种参数格式, * | _all | name1, name2, ...

过滤配置参数的返回结果:

```
GET http://127.0.0.1:9200/secisland/_settings/name=index.number_*
```

name=index.number_* 设置将只返回 number_of_replicas, number_of_shards 两个参数详情。

2.4.3 索引分析

索引分析 (analysis) 是这样一个过程: 首先, 把一个文本块分析成一个个单独的词 (term), 为了后面的倒排索引做准备。然后标准化这些词为标准形式, 提高它们的“可搜索性”。这些工作是分析器 (analyzers) 完成的。一个分析器 (analyzers) 是一个组合, 用于将三个功能放到一起:

- ❑ 字符过滤器: 字符串经过字符过滤器 (character filter) 处理, 它们的工作是在标记化之前处理字符串。字符过滤器能够去除 HTML 标记, 或者转换 “&” 为 “and”。
- ❑ 分词器: 分词器 (tokenizer) 被标记化成独立的词。一个简单的分词器 (tokenizer) 可以根据空格或逗号将单词分开。
- ❑ 标记过滤器: 每个词都通过所有标记过滤 (token filters) 处理, 它可以修改词 (例如将 “Quick” 转为小写), 去掉词 (例如连接词像 “a”、“and”、“the” 等), 或者增加词 (例如同义词像 “jump” 和 “leap”)。

Elasticsearch 提供很多内置的字符过滤器, 分词器和标记过滤器。这些可以组合起来创

建自定义的分析器以应对不同的需求。

测试分析器:

请求: POST 127.0.0.1:9200/_analyze

```
{
  "analyzer" : "standard",
  "text" : "this is a test"
}
```

该结果将返回“this is a test”使用 standard 分析器后词的解析情况。

在该分析器下,将会分析成 this, is, a, test 四个词。

自定义分析器:

请求: POST 127.0.0.1:9200/_analyze

```
{
  "tokenizer" : "keyword",
  "token_filters" : ["lowercase"],
  "char_filters" : ["html_strip"],
  "text" : "this is a <b>test</b>"
}
```

使用 keyword 分词器、lowercase 分词过滤、字符过滤器是 html_strip, 这 3 部分构成一个分词器。

上面示例返回分词结果是 this is a test, 其中 html_strip 过滤掉了 html 字符。

也可以指定索引进行分词。URL 格式如下:

http://127.0.0.1:9200/secisland/_analyze

索引详情如下: 如果想获取分析器分析的更多细节, 设置 explain 属性为 true (默认是 false), 将输出分词器的分词详情。请求格式如下:

请求: POST http://127.0.0.1:9200/secisland_analyze

参数:

```
{
  "tokenizer" : "standard",
  "token_filters" : ["snowball"],
  "text" : "detailed output",
  "explain" : true,
  "attributes" : ["keyword"]
}
```

返回值如下:

```
{
  "detail" : {
    "custom_analyzer" : true,
    "charfilters" : [],
    "tokenizer" : {
```


判断索引模板是否存在：

```
HEAD http://127.0.0.1:9200/_template/template_1
```

4. 多个模板匹配

有这样一种情况：`template_1`、`template_2` 两个模板，使用 `te*` 会匹配 2 个模板，最后合并两个模板的配置。如果配置重复，这时应该设置 `order` 属性，`order` 是从 0 开始的数字，先匹配 `order` 数字小的，再匹配数字大的，如果有相同的属性配置，后匹配的会覆盖之前的配置。

2.4.5 复制配置



复制配置功能是实验性的，在未来的版本可能会改变。

如果使用共享文件系统，可以使用复制配置来选择索引数据保存在磁盘上的位置，以及 Elasticsearch 如何在索引副本分片上重复操作。

为了充分利用 `index.data_path` 和 `index.shadow_replicas` 设置，要让 Elasticsearch 使用相同数据目录为多个实例服务，需要在 `elasticsearch.yml` 中设置 `node.add_id_to_custom_path` 为 `false`：

```
node.add_id_to_custom_path: false
```

可以在 `elasticsearch.yml` 中设置 `path.shared_data` 权限管理标明自定义索引的位置，以便应用正确的权限：

```
path.shared_data: /opt/data
```

这意味着 Elasticsearch 可以读写 `path.shared_data` 设置中所有子目录中的文档。

可以在自定义数据路径中创建索引，每个节点使用这个路径来存储数据：

```
PUT http://127.0.0.1:9200/secisland
{
  "index" : { "number_of_shards" : 1, "number_of_replicas" : 4,
    "data_path": "/opt/data/secisland", "shadow_replicas": true
  }
}
```

索引创建时，`index.shadow_replicas` 设置为 “`true`” 不会在任何副本分片上重复文档操作，反而，会不停的刷新。

下面是可以使用设置更新接口修改的设置列表：

- ❑ `index.data_path`——字符型索引数据使用的路径。Elasticsearch 默认附加节点序号到路径中，确保相同机器上的多重实例不会共享数据目录。
- ❑ `index.shadow_replicas`——布尔值，指示索引是否应该使用副本。默认为 `false`。

- ❑ `index.shared_filesystem`——布尔值，指示索引使用共享文件系统。如果 `index.shadow_replicas` 设置为 `true`，默认值为 `true`，其他情况下的默认值为 `false`。
- ❑ `index.shared_filesystem.recover_on_any_node`——布尔值，指示索引的主分片是否可以恢复到集群中的任何节点。默认值为 `false`。

2.4.6 重建索引

重建索引是 2.3.0 新增加的接口。这个接口是实验性质的，在未来有可能会改变。

重建索引的最基本的功能是拷贝文件从一个索引到另一个索引，例如：

请求：POST `http://127.0.0.1:9200/_reindex`

```
{
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland"}
}
```

返回值如下：

```
{
  "took" : 639,
  "updated": 112,
  "batches": 130,
  "version_conflicts": 0,
  "failures" : [ ],
  "created": 12344
}
```

参数说明：

`took`：从开始到结束的整个操作的毫秒数。

`updated`：已成功更新的文档数。

`created`：成功创建的文档数。

`batches`：从重建索引拉回的滚动响应的数量。

`version_conflicts`：重建索引中版本冲突数的数量。

`failures`：所有索引失败的数组。如果这是非空的，则请求将被中止。

由于 `_reindex` 是获取源索引的快照，而且目标索引是不同的索引，所以基本上不太可能产生冲突。在接口参数中可以增加 `dest` 来进行乐观并发控制。如果 `version_type` 设置为 `internal` 会导致 Elasticsearch 盲目转储文件到目标索引，任何具有相同类型和 ID 的文档将被重写。例如：

请求：POST `http://127.0.0.1:9200/_reindex`

```
{
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland", "version_type": "internal"}
}
```

如果设置 `version_type` 为 `external` 将会导致 Elasticsearch 保护源索引的版本，如果在目标索引中有一个比源索引旧的版本，则会更新文档。对于源文件中丢失的文档在目标中也会被创建。

请求: `POST http://127.0.0.1:9200/_reindex`

```
{
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland", "version_type": "external"}
}
```

设置 `op_type` 为 `create` 将导致 `_reindex` 在目标索引中仅创建丢失的文件。所有现有的文件将导致版本冲突。

请求: `POST http://127.0.0.1:9200/_reindex`

```
{
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland", "op_type": "create"}
}
```

正常情况下当发生冲突的时候 `_reindex` 过程将被终止，可以在请求体中设置 `"conflicts": "proceed"`，可以只进行计算：

请求: `POST http://127.0.0.1:9200/_reindex`

```
{
  "conflicts": "proceed",
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland", "op_type": "create"}
}
```

可以通过向源添加一个类型或者增加一个查询来限制文档的数量，比如只复制用户名为 `kimchy` 的文档：

请求: `POST http://127.0.0.1:9200/_reindex`

```
{
  "source": {
    "index": "secisland",
    "type": "secilog",
    "query": {"term": {"user": "kimchy"}}
  },
  "dest": {"index": "new_secisland"}
}
```

在请求接口中可以列出源索引和类型，可以在一个接口中复制多个源。例如下面的例子将在 `secisland` 和 `blog` 索引中的 `secilog` 和 `post` 类型中拷贝数据，这包括 `secisland` 索引中的 `"secilog"` 和 `"post"` 类型，也包括 `blog` 索引中的 `"secilog"` 和 `"post"` 类型。如果需要更具体的文档可以使用查询。当 `id` 产生冲突的时候是没有办法处理的，因为执行的顺序是随

机的, 所以目标索引将无法确定应该保存哪些文档:

请求: POST http://127.0.0.1:9200/_reindex

```
{
  "source": {"index": ["secisland", "blog"], "type": ["secilog", "post"]},
  "dest": {"index": "all_together"}
}
```

也可以通过设置大小来限制处理文档的数量。这只会复制一个文件到 new_secisland 索引中:

请求: POST http://127.0.0.1:9200/_reindex

```
{
  "size": 1,
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland"}
}
```

如果你想要复制特定的文档, 可以使用排序。排序会降低效率, 但在某些情况下, 它是有意义的。如果可能的话, 可以选择性地查询来确定复制的大小和排序。下面将从 secisland 索引中复制 10000 文档到 new_secisland 中:

请求: POST http://127.0.0.1:9200/_reindex

```
{
  "size": 10000,
  "source": {"index": "secisland", "sort": { "date": "desc" }},
  "dest": {"index": "new_secisland"}
}
```

_reindex 同时支持使用脚本来修改文档。例如:

请求: POST http://127.0.0.1:9200/_reindex

```
{
  "source": {"index": "secisland"},
  "dest": {"index": "new_secisland", "version_type": "external"}
  "script": {
    "internal": "if (ctx._source.foo == 'bar') {ctx._version++; ctx._source.remove('foo')}"}
}
```

可以修改的元字段包括 _id、_type、_index、_version、_routing、_parent、_timestamp、_ttl。所以用脚本复制能力非常强大。

默认情况下, 如果 _reindex 可以设置文档路由来进行路由保存, 除非是通过脚本进行了改变。你可以设置 dest 参数来进行路由设置:

- ❑ keep: 设置在匹配的每一个路由上发送请求的路由。这个是默认设置。
- ❑ discard: 为每一个匹配发送的请求设置为空。
- ❑ =<some text>: 设置为每一个匹配的文本请求发送路由上的路由。例如, 你可以使用

以下要求复制所有文件从源索引为“company”等于“cat”的查询中复制到目标索引，目标索引的路由设置为 cat：

请求：POST http://127.0.0.1:9200/_reindex

```
{
  "source": {"index": "source", "query": {"match": {"company": "cat"}}},
  "dest": {"index": "dest", "routing": "=cat"}
}
```

默认情况下 _reindex 每次处理的大小为 100。可以在源参数中用 size 参数来进行修改：

请求：POST http://127.0.0.1:9200/_reindex

```
{
  "source": {"index": "source", "size": 1500},
  "dest": {"index": "dest"}
}
```

URL 参数：_reindex 接口除了接收标准的参数例如 pretty，还支持 refresh, wait_for_completion、consistency、timeout 参数。

在 URL 参数中发送 refresh 会导致所有写入请求的索引被刷新。与索引接口中的 refresh 参数不同，只有接收到新数据的分片会进行刷新。

如果请求包含 wait_for_completion = false，Elasticsearch 将进行执行前检查后启动请求，然后返回一个 task 可用于任务 API 取消或得到任务的状态，现在一旦请求完成任务就不见了，找任务的最终结果的唯一地方是在 Elasticsearch 日志文件中，这个问题将会在未来的版本中修复。

consistency 控制每次写请求必须多少份分片被响应。

timeout 控制每个写请求等待可用的分片的时间。

任务查看：

请求：GET http://127.0.0.1:9200/_tasks/?pretty&detailed=true&actions=*reindex

返回值类似于：

```
{
  "nodes" : {
    "r1A2WoRbTwKZ516z6NEs5A" : {
      "name" : "Tyrannus",
      "transport_address" : "127.0.0.1:9300",
      "host" : "127.0.0.1",
      "ip" : "127.0.0.1:9300",
      "attributes" : {
        "testattr" : "test",
        "portsfile" : "true"
      },
      "tasks" : {
        "r1A2WoRbTwKZ516z6NEs5A:36619" : {
          "node" : "r1A2WoRbTwKZ516z6NEs5A",
```

```

    "id" : 36619,
    "type" : "transport",
    "action" : "indices:data/write/reindex",
    "status" : {
      "total" : 6154,
      "updated" : 3500,
      "created" : 0,
      "deleted" : 0,
      "batches" : 36,
      "version_conflicts" : 0,
      "noops" : 0
    },
    "description" : ""
  }
}
}
}
}

```

`_reindex` 可以用来建立索引的时候同时也可以修改列的名称, 例如, 源索引类型为:

请求: `POST http://127.0.0.1:9200/secisland/secilog/1?refresh&pretty`

```

{
  "text": "words words", "flag": "foo"
}

```

可以通过 `_reindex` 来修改列的名称, 例如:

请求: `POST http://127.0.0.1:9200/_reindex?pretty`

```

{
  "source": {"index": "secisland"},
  "dest": {"index": "secisland2"},
  "script": {"inline": "ctx._source.tag = ctx._source.remove(\"flag\")"}
}

```

然后我们看一下 `secisland2` 的结构, 可以看出 `flag` 字段名称修改成了 `tag`:

请求: `GET http://127.0.0.1:9200/secisland2/secilog/1?pretty`

```

{
  "text": "words words", "tag": "foo"
}

```

2.5 索引监控

在 Elasticsearch 中, 系统提供了接口来监控索引的状态, 包括索引的统计信息、碎片信息、恢复的状态和分片的信息, 利用这些接口可以随时监控系统索引的状态。

2.5.1 索引统计

索引统计接口提供索引中不同内容的统计数据 (其中的大多数统计数据也可以从节点级

别范围取得)。

获取所有聚合以及索引的统计数据:

请求: GET http://127.0.0.1:9200/_stats

获取指定索引的统计数据:

请求: GET http://127.0.0.1:9200/index1,index2/_stats

默认情况返回所有统计数据,也可以在 URL 中指定需要返回的特定统计数据。指定数据如表 2-1 所示。

表 2-1 统计数据选项

指定统计数据	说明
docs	文档 / 删除文档 (没有合并的文档) 的数量。注意,受索引刷新的影响
store	索引的大小
indexing	索引统计数据,可以结合用逗号分隔的类型列表来提供文档类型级别的统计数据
get	获取统计数据,包含缺失统计
search	搜索统计数据。可以通过添加额外的 groups 参数 (搜索操作可以关联一个或多个分组) 包含自定义分组的统计数据。groups 参数接受逗号分隔的组名列表。使用 _all 来返回所有分组的统计数据
completion	完成建议统计数据
fielddata	字段数据统计数据
flush	冲洗统计数据
merge	混合统计数据
request_cache	分片请求缓存统计数据
refresh	刷新统计数据
suggest	建议统计数据
translog	事务日志统计数据

一些统计数据可以作用在字段粒度上,接受逗号分隔的字段列表。默认包含所有字段,如下所示:

□ fields——包含在统计数据中的字段列表。用作默认列表,除非提供了更明确的列表。

□ completion_fields——包含在完成建议统计数据中的字段列表。

□ fielddata_fields——包含在字段数据统计数据中的字段列表。

举一些例子:

□ 获取所有索引的混合和刷新统计数据:

请求: GET http://127.0.0.1:9200/_stats/merge,refresh

□ 获取名为 secisland 索引中类型为 type1 和 type2 的文档统计数据:

请求: GET http://127.0.0.1:9200/secisland/_stats/indexing?types=type1,type2

□ 获取分组 group1 和 group2 的搜索统计数据:

请求: GET http://127.0.0.1:9200/_stats/search?groups=group1,group2

返回的统计数据在索引级别发生聚合，生成名为 **primaries** 和 **total** 的聚合。其中，**primaries** 仅包含主分片的值，**total** 是主分片和从分片的累计值。

为了获取分片级别统计数据，需要设置 **level** 参数为 **shards**。

注意，当分片在集群中移动的时候，它们的统计数据会被清除，视作它们在其他节点中被创建。另一方面，即使分片“离开”了一个节点，那个节点仍然会保存分片之前的统计数据。

2.5.2 索引分片

提供 Lucene 索引所在的分片信息。可以用来提供分片和索引的更多统计信息，可能是优化信息，删除的“垃圾”数据，等等。

参数包括特定的索引，多个索引或者所有索引的分片请求如下：

- ❑ GET http://127.0.0.1:9200/secisland/_segments
- ❑ GET http://127.0.0.1:9200/secisland1,secisland2/_segments
- ❑ GET http://127.0.0.1:9200/_segments

返回值：

```
{
  ...
  "_3": {
    "generation": 3,
    "num_docs": 1121,
    "deleted_docs": 53,
    "size_in_bytes": 228288,
    "memory_in_bytes": 3211,
    "committed": true,
    "search": true,
    "version": "4.6",
    "compound": true
  }
  ...
}
```

其中的参数见表 2-2。

表 2-2 索引分片信息

参数	说明
_3	JSON 文档的键名，代表分片的名称。这个名用来生成文档名：分片目录中所有以分片名开头的文档属于这个分片
generation	需要写新的分片时生成的一个数字，基本上是递增的。分片名从这个生成的数字派生出来
num_docs	存储在分片中没被删除的文档数量
deleted_docs	存储在分片中被删除的文档数量。如果这个数字大于 0 也是没有问题的，磁盘空间会在分片融合的时候被回收

(续)

参数	说明
size_in_bytes	用字节表示分片使用的磁盘空间数量
memory_in_bytes	分片需要在内存中存储一些数据使搜索性能更加高效。这个数字表示用于这个目的字节数量。如果返回的值为 -1，表示 Elasticsearch 无法计算这个值
committed	表示分片在磁盘上是否同步。提交的分片会在硬重启中存活下来。如果值为 false 也不需要担心，未提交的分片数据也会存储在事务日志中，在 Elasticsearch 下一次启动时可以重做修改
search	分片是否可以进行搜索。如果值为 false，可能意味着分片已经被写入磁盘但是没有经过刷新使之可以进行搜索
version	用来写这个分片的 Lucene 版本
compound	分片是否存储在符合文件中。如果值为 true，意味着 Lucene 将分片中的所有文档融合为一个用来保存文档的描述符

2.5.3 索引恢复

索引恢复接口提供正在进行恢复的索引分片信息。可以报告指定索引或者集群范围的恢复状态。

例如，获取 secisland1 和 secisland2 两个索引的恢复信息：

```
GET http://127.0.0.1:9200/secisland1,secisland2/_recovery
```

去掉索引名可以查看集群范围的恢复状态：

```
GET http://127.0.0.1:9200/_recovery?pretty&human
```

选项列表：

☐ detailed——显示详细的视图。主要用来查看物理索引文件的恢复。默认为 false。

☐ active_only——显示那些现在正在进行的恢复。默认为 false。

输出字段描述如表 2-3 所示。

表 2-3 索引恢复输出字段

字段	说明
id	分片 ID
type	恢复类型：包括存储、快照、复制、迁移
stage	恢复阶段：包括初始化（恢复没有开始）、索引（读取索引元字段并且从源到目的地复制字节）、开始（启动恢复；开启使用的索引）、事务日志（重做事务日志）、完成（清理）、结束
primary	如果分片是主分片，值为 true；分片是从分片，值为 false
start_time	恢复开始的时间戳
stop_time	恢复结束的时间戳
total_time_in_millis	以毫秒来表示恢复分片的整个时间

(续)

字段	说明
source	恢复源：如果从快照中恢复，描述备份仓库；其他情况，描述源节点
target	目标节点
index	物理索引恢复的统计数据
translog	事务日志恢复的统计数据
start	打开和启动索引的时间统计数据

2.5.4 索引分片存储

提供索引分片副本的存储信息。存储信息报告分片副本存在的节点、分片副本版本、指示分片副本最近的状态以及在开启分片索引时遭遇的任何异常。

默认情况下，只列出至少有一项未分配副本的分片的存储信息。

端点包括特定的索引、多个索引或者所有索引的分片：

- ☐ GET http://127.0.0.1:9200/secisland/_shard_stores
- ☐ GET http://127.0.0.1:9200/secisland1,secisland2/_shard_stores
- ☐ GET http://127.0.0.1:9200/_shard_stores

列出存储信息的分片范围可以通过 status 参数进行修改。默认是 yellow 和 red。使用 green 参数来列出所有指定副本分片的存储信息：GET http://127.0.0.1:9200/_shard_stores?status=green。

2.6 状态管理

在 Elasticsearch 中还有一些和索引相关的重要接口需要介绍，这些接口包括清除索引缓存、索引刷新、冲洗索引、合并索引接口等，下面分别介绍。

2.6.1 清除缓存

清除缓存接口可以清除所有缓存或者关联一个或更多索引的特定缓存：

POST http://127.0.0.1:9200/secisland/_cache/clear

接口默认清理所有缓存，可以明确设置 query、fielddata 和 request 来清理特定缓存。

所有关联特定字段的缓存也可以被清理，通过逗号分隔的相关字段列表来指定 fields 参数。

2.6.2 索引刷新

刷新接口可以明确地刷新一个或多个索引，使之前最后一次刷新之后的所有操作被执

行。(接近)实时能力取决于使用的搜索引擎。例如,内部的一个请求刷新被调用,但是默认刷新是一个周期性的安排。

```
POST http://127.0.0.1:9200/secisland/_refresh
```

刷新接口可以通过一条请求应用到多个索引,或者所有索引:

```
POST http://127.0.0.1:9200/secisland,elasticsearch/_refresh
```

```
POST http://127.0.0.1:9200/_refresh
```

2.6.3 冲洗

冲洗(flush)接口可以通过接口冲洗一个或多个索引。索引主要通过执行冲洗将数据保存到索引存储并且清除内部事务日志,以此来释放索引的内存空间。默认的,Elasticsearch使用内存启发式算法来自动触发冲洗操作的请求来清理内存:

```
POST http://127.0.0.1:9200/secisland/_flush
```

2.6.4 合并索引

合并接口可以强制合并一个或更多索引。合并分片数量和每个分片保存的 Lucene 索引。强制合并可以通过合并来减少分片数量。

调用会被阻塞直到合并完成。如果 http 连接丢失,请求会在后台继续执行,任何新的请求都会被阻塞直到前一个强制合并完成。

```
POST http://127.0.0.1:9200/secisland/_forcemerge
```

合并接口接受请求参数如下所示:

❑ **max_num_segments**——用于合并的分片数量。为了充分合并索引,设置它的值为 1。默认简单地检查是否需要执行合并,如果是,执行合并。

❑ **only_expunge_deletes**——合并过程是否只删除分片中被删除的文档。在 Lucene 中,文档不会从分片中删除,只是标记为删除。通过执行分片合并,一个不包含这些被删除的文档的新分片会被创建。这个标识可以只合并拥有删除文档的分片。默认值为 false。注意 `index.merge.policy.expunge_deletes_allowed` 阈值不会被覆盖。

❑ **flush**——强制合并之后是否执行冲洗。默认为 true。

合并接口可以通过单次应用到多个索引,或者所有索引:

```
POST http://127.0.0.1:9200/secisland,elasticsearch/_forcemerge
```

```
POST http://127.0.0.1:9200/_forcemerge
```

2.7 文档管理

文档是具体的数据，一个文档有点像数据库中的一条记录，文档必须包含在一个索引中。

2.7.1 增加文档

新增文档是把一条新的文档增加到索引中，使之能够进行搜索，文档的格式是 JSON 格式。注意，在 Elasticsearch 中如果有相同 ID 的文档存在，则更新此文档。例如我们在索引 `secilog` 中增加一条文档：

请求：PUT `http://127.0.0.1:9200/secisland/secilog/1`

参数：

```
{
  "collect_type" : "syslog",
  "collect_date" : "2016-01-11T09:32:12",
  "message" : "Failed password for root from 192.168.21.1 port 50790 ssh2"
}
```

返回值：

```
{
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 1,
  "_shards": {"total": 2, "successful": 1, "failed": 0},
  "created": true
}
```

返回值中的 `_shards` 提供了索引创建的过程信息，参数如下所示：

- ❑ `total`——文档被创建的时候，在多少个分片中进行了操作，包括主分片和副本分片。
- ❑ `successful`——成功建立索引分片的数量，当创建成功后，成功创建索引分片的数量最少是 1。
- ❑ `failed`——失败建立索引分片的数量。

1. 自动创建索引

当创建文档的时候，如果索引不存在，则会自动创建该索引。自动创建的索引会自动映射每个字段的类型。自动创建字段类型是非常灵活的，新的字段类型将会自动匹配字段对象的类型。比如字符串类型，日期类型。可以通过配置文件设置 `action.auto_create_index` 为 `false` 在所有节点的配置文件中禁用自动创建索引。自动映射的字段类型可以通过配置文件设置 `index.mapper.dynamic` 为 `false` 禁用。自动创建索引可以通过模板设置索引名称，例如：可以设置 `action.auto_create_index` 为 `+aaa*,-bbb*,+ccc*,*(-`（+ 表示准许，- 表示禁止）。

2. 版本号

每个文档都有一个版本号，版本号的具体值放在创建索引的返回值中（"_version":）。通过版本号参数可以达到并发控制的效果。当在操作文档的过程中指定版本号，如果和版本号不一致的时候操作会被拒绝。版本号常用在对事务的处理中。例如，更新刚才创建的文档：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1?version=2&pretty

参数：

```
{
  "message" : "elasticsearch now has versioning support!"
}
```

返回值为更新失败：

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "version_conflict_engine_exception",
      "reason" : "[secilog][1]: version conflict, current [-1], provided [2]",
      "index" : "secisland",
      "shard" : "2"
    } ],
    "type" : "version_conflict_engine_exception",
    "reason" : "[secilog][1]: version conflict, current [-1], provided [2]",
    "index" : "secisland",
    "shard" : "2"
  },
  "status" : 409
}
```



注意 版本号是实时更新的，不会存在缓存现象。当操作的时候不指定版本号，则系统不会对版本号是否一致进行检查。

默认情况下对文档的操作版本号从 1 开始递增，包括修改文档和删除文档。当然版本号还可以从外部获取，比如从数据库中获取，要启用此功能，`version_type` 应设置为 `external`，这个值必须是一个大于 0 小于 $9.2e+18$ 的数字。当使用外部版本号来代替自动生成的版本号时，在操作文档的时候，系统通过对比参数中的版本号是否大于文档中的版本号来做判断，当参数中的版本号大于系统中的版本号，则执行此操作，并更新版本号。反之则拒绝操作（包括小于或者等于）。

版本号同时产生了一个比较实用的功能，只要版本号从源数据库中使用，在异步索引操作的时候就不需要对源数据库的变化执行严格排序。任何操作都只会对最新的版本号起作用，不管这个版本号是内部的还是从外部获取的。

3. 操作类型

系统同时支持通过 `op_type=create` 参数强制命令执行创建操作，只有系统中不存在此文档的时候才会创建成功。如果不指定此操作类型，如果存在此文档，则会更新此文档。例如再次创建文档：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1?op_type=create&pretty

参数：

```
{
  "collect_type" : "syslog",
  "collect_date" : "2016-01-11T09:32:12",
  "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

返回值，表示创建失败：

```
{
  "error" : {
    "root_cause" : [ {
      "type" : "document_already_exists_exception",
      "reason" : "[secilog][1]: document already exists",
      "index" : "secisland",
      "shard" : "3"
    } ],
    "type" : "document_already_exists_exception",
    "reason" : "[secilog][1]: document already exists",
    "index" : "secisland",
    "shard" : "3"
  },
  "status" : 409
}
```

当不指定 `op_type=create` 时候，则更新此文档。

创建操作的另一个写法为：http://127.0.0.1:9200/secisland/secilog/1/_create?pretty

4. 自动创建 ID

当创建文档的时候，如果不指定 ID，系统会自动创建 ID。自动生成的 ID 是一个不会重复的随机数。例如：

请求：POST http://127.0.0.1:9200/secisland/secilog/?op_type=create&pretty

参数：

```
{
  "collect_type" : "syslog",
  "collect_date" : "2016-01-11T09:32:12",
  "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

返回值：


```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "AVLOV99W6rG7Qqt6i_gk",
  "_version" : 1,
  "_shards" : { "total" : 2, "successful" : 1, "failed" : 0 },
  "created" : true
}
```

从中可以看出 `_id` 自动生成了一个随机数。

5. 分片选择

默认情况下，分片的选择是通过 ID 的散列值进行控制。这个只可以通过 `router` 参数进行手动的控制。可以在每个操作的基础上直接通过哈希函数的值来指定分片的选择。例如：

请求：POST `http://127.0.0.1:9200/secisland/secilog/?routing=secisland&pretty`

在上面的例子中，分片的选择是通过指定 `routing=secisland` 参数的哈希值来确定的。

6. 其他说明

- 分布式：索引操作主要是针对主节点的分片进行，当主节点完成索引操作后，如果有副本节点，则分发到副本中。
- 一致性：为了防止当网络出现问题时写入不一致，系统只有在有效节点的数量大于一定数量的时候生效（总结点数 $/2+1$ ），该值可以通过 `action.write_consistency` 参数进行修改。
- 刷新：更新的时候可以指定 `refresh` 参数为 `true` 来立即刷新所有的副本，当 `refresh` 设置为 `true` 的时候，系统做了充分的优化，不会对系统产生任何影响，需要注意的是查询操作 `refresh` 参数没有任何的意义。
- 空操作：当文档内容没有任何改变的时候，更新文档操作也会生效，具体体现在版本号会发生变化。如果不希望此情况发生，在更新的时候指定 `detect_noop` 为 `true`。这个参数在创建索引的时候无效。
- 超时：默认情况下系统的超时时间是 1 分钟。可以通过设置 `timeout` 来修改超时的时间，例如 `timeout=5m`，表示超时的时间是 5 分钟。

2.7.2 更新删除文档

Elasticsearch 的更新文档 API 准许通过脚本操作来更新文档。更新操作从索引中获取文档，执行脚本，然后获得返回结果。它使用版本号来控制文档获取或者重建索引。



在 Elasticsearch 中的更新操作是完全重新索引文件。

我们新建一个文档。

请求: PUT http://127.0.0.1:9200/secisland/secilog/1?pretty

参数:

```
{
  "counter" : 1,
  "tags" : ["red"]
}
```

1. 脚本开启功能

在最新版本的 Elasticsearch 中, 基于安全考虑 (如果用不到, 请保持禁用), 默认禁用了动态脚本功能。如果被禁用, 在使用脚本的时候则报以下的错误:

```
scripts of type [inline], operation [update] and lang [groovy] are disabled
```

可以用以下方式完全开启动态脚本功能, 在 config/elasticsearch.yml 文件最后添加以下代码:

```
script.inline: on
script.indexed: on
script.file: on
```

配置后, 重启 Elasticsearch。

2. 用脚本更新文稿

下面我们用脚本来更新此文档。

请求: POST http://127.0.0.1:9200/secisland/secilog/1/_update?pretty

参数:

```
{
  "script" : {
    "inline": "ctx._source.counter += count",
    "params" : {"count" : 4}
  }
}
```

执行完后, 我们再查询一下文档内容, 可以发现 counter 的值为 5:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 5,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red" ]
  }
}
```

再看下面的更新操作。

请求: POST http://127.0.0.1:9200/secisland/secilog/1/_update?pretty

参数:

```
{
  "script" : {
    "inline": "ctx._source.tags += tag",
    "params" : {"tag" : "blue"}
  }
}
```

返回值表示更新成功,我们看一下 _version 为 6,比刚才的值增加了 1:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 6,
  "_shards" : {"total" : 2, "successful" : 1, "failed" : 0}
}
```

然后我们再查询一下文档内容:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 6,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ]
  }
}
```

在脚本中除了 _source 外其他内置参数也可以使用,例如 _index、_type、_id、_version、_routing、_parent、_timestamp、_ttl。

下面我们通过脚本增加一列。

请求: POST http://127.0.0.1:9200/secisland/secilog/1/_update?pretty

参数:

```
{
  "script" : "ctx._source.name_of_new_field = \"value_of_new_field\""
}
```

然后查询此文档:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 7,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ],
    "name_of_new_field" : "value_of_new_field"
  }
}
```

```

    "found" : true,
    "_source" : {
      "counter" : 5,
      "tags" : [ "red", "blue" ],
      "name_of_new_field" : "value_of_new_field"
    }
  }
}

```

从中可以看出，文档中又增加了一列。

如果删除一行，请求和刚才的一样，参数变为：

```

{
  "script" : "ctx._source.remove(\"name_of_new_field\")"
}

```

甚至可以通过表达式来判断做某些事情。例如：下面的示例将删除 tag 字段包含 blue 的文件。

请求参数：

```

{
  "script" : {
    "inline": "ctx._source.tags.contains(tag) ? ctx.op = \"delete\" : ctx.op = \"none\"",
    "params" : { "tag" : "blue" }
  }
}

```

部分文档更新：该更新接口还支持更新部分文档，将文档合并到现有文档中（简单的递归合并、对象的内部合并、替换核心的“键/值”和数组）。例如：

```

{
  "doc" : { "name" : "new_name" }
}

```

更新后，可以发现文档中多了一列 name：

```

{
  "_index" : "test",
  "_type" : "type1",
  "_id" : "1",
  "_version" : 23,
  "found" : true,
  "_source" : {
    "counter" : 5,
    "tags" : [ "red", "blue" ],
    "name" : "new_name"
  }
}

```

当文档指定的值与现有的 _source 合并，当新的文档和老的文档不一致的时候，文档将会被重新建立索引。当新旧文档一样的时候，则不进行重建索引的操作。可以通过设置 detect_noop 为 false，让任何情况下都重新建立索引，例如下面的更新操作：

```
{
  "doc" : { "name" : "new_name",
    "detect_noop": false
  }
}
```

删除文档相对比较简单。

请求: DELETE http://127.0.0.1:9200/secisland/secilog/1

返回值:

```
{
  "found": true,
  "_index": "secisland",
  "_type": "secilog",
  "_id": "1",
  "_version": 24,
  "_shards": { "total": 2, "successful": 1, "failed": 0 }
}
```

则表示删除了此文档。

2.7.3 查询文档

Elasticsearch 查询文档 API 准许用户通过文档的 ID 来查询具体的某一个文档, 例如下面查询索引为 secilog, type 为 log, id 为 1 的文档:

请求: GET http://127.0.0.1:9200/secisland/secilog/1?pretty

返回值:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "collect_type" : "syslog",
    "collect_date" : "2016-01-11T09:32:12",
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
  }
}
```

从返回的内容中可以有很多有用的信息, 当然也包括原始的文档信息, 放在 _source 字段中。

默认情况下, 查询获得的数据接口是实时的, 并且不受索引的刷新率影响, 为了禁用实时性, 可以将参数 realtime 设置为 false, 或全局设置 action.get.realtime 为 false。



在查询中, 中间的 _type 是可选的, 当不指定具体 type 的时候, 可以用 _all 来代替。

默认情况下, 查询操作会返回 `_source` 字段, 当然 `_source` 可以被禁用。例如:

请求: `GET http://127.0.0.1:9200/secisland/secilog/1?_source=false&pretty`

返回值:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 2,
  "found" : true
}
```

当然如果你想获取 `source` 中的一部分内容, 可以用 `_source_include` 或者 `_source_exclude` 来包含或者过滤其中的某些字段, 例如:

请求: `GET http://127.0.0.1:9200/secisland/secilog/1?_source_include=message&pretty`

返回值:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
  }
}
```

当一个文档内容非常多的时候, 用包含或者过滤可以减少很多的网络负担。如果有多个, 可以用逗号分开, 或者用 `*` 通配符。例如:

请求: `GET http://127.0.0.1:9200/secisland/secilog/1?_source_include=message,collect_date&pretty`

返回值:


```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "_source" : {
    "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2",
    "collect_date" : "2016-01-11T09:32:12"
  }
}
```

通过 `fields` 字段过滤, 可以从存储中查询一组字段, 例如:

请求: `GET http://127.0.0.1:9200/secisland/secilog/1?fields=message,collect_date&pretty`

返回值:

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 2,
  "found" : true,
  "fields" : {
    "message" : [ "Failed password for root from 192.168.21.2 port 50790 ssh2" ],
    "collect_date" : [ "2016-01-11T09:32:12" ]
  }
}
```

 **提示** 从返回值可以看出，返回的字段是数组类型的，但 `_routing` 字段和 `_parent` 字段是没有数组返回的。只有子字段可以从 `fields` 中进行查询，对象数据是不生效的。

如果建立索引后还没有来得及刷新，查询得到的内容是事务的日志。但有些字段只有在索引的时候才会产生，当访问这些字段的时候，系统会抛出一个异常。可以通过设置 `ignore_errors_on_generated_fields=true` 来忽略这些字段。

1. 只获取文档内容

可以通过 `{/index}/{/type}/{/id}/_source` 的方式只获取文档内容，例如：

请求：GET `http://127.0.0.1:9200/secisland/secilog/1/_source?pretty`

返回值：

```
{
  "collect_type" : "syslog",
  "collect_date" : "2016-01-11T09:32:12",
  "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}
```

同理，这种方式的查询也可以通过之前的过滤方式来选择具体的字段。

2. 分片选择 (routing)

可以在查询的时候指定路由选择 (routing)，当路由不存在的时候，返回为空值，此实例是事先做了路由的操作，例如：

请求：GET `http://127.0.0.1:9200/secisland/secilog/1?routing=secisland&pretty`

返回值：

```
{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 1,
  "_routing" : "secisland",
  "found" : true,

```

```

    "_source" : {
      "collect_type" : "syslog",
      "collect_date" : "2016-01-11T09:32:12",
      "message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
    }
  }
}

```

3. 查询参数

通过参数控制，查询的时候可以指定查询是在主节点上查询还是在副本节点上查询。参数有：

- ❑ `_primary`：在主节点进行查询。
- ❑ `_local`：尽可能在本地节点上进行查询。
- ❑ `refresh`：可以设置为 `true`，使之在搜索操作前刷新相关的分片保证可以及时查询到。但这个参数会消耗系统的资源，除非有必要，正常情况下不需要设置。例如：

例如：GET `http://127.0.0.1:9200/secisland/secilog/1?_primary&refresh=true`

2.7.4 多文档操作

在 Elasticsearch 对文档的操作中，之前介绍的都是对单个文档进行操作，其实 Elasticsearch 可以对多个文档同时操作。下面介绍多文档查询 (bulk)。

1. 多文档查询

多文档查询可以在同一个接口中查询多个文档，可以分别指定 `index`、`type`、`id` 来进行多个文档的查询。响应包括所有查询到的文档数组，每个元素在结构上类似于单个文档查询。下面举例说明。

请求：POST `http://127.0.0.1:9200/_mget?pretty`

参数：

```

{
  "docs" : [
    { "_index" : "secisland", "_type" : "secilog", "_id" : "1" },
    { "_index" : "secisland", "_type" : "secilog", "_id" : "2" }
  ]
}

```

返回值：

```

{
  "docs" : [ {
    "_index" : "secisland",
    "_type" : "secilog",
    "_id" : "1",
    "_version" : 3,
    "found" : true,
    "_source" : {

```

```

"collect_type" : "syslog",
"collect_date" : "2016-01-11T09:32:12",
"message" : "Failed password for root from 192.168.21.2 port 50790 ssh2"
}, {
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "2",
  "_version" : 1,
  "found" : true,
  "_source" : {
    "collect_type" : "syslog",
    "collect_date" : "2016-01-12T09:32:12",
    "message" : "secisland mget test!"
  }
}
]

```

从中可以看出，一次查询了两个文档。

在查询的时候，index、type 可以在 URL 中直接填写。例如下面两个请求和之前的是等价的。

请求：POST http://127.0.0.1:9200/secisland/_mget?pretty

参数：

```

{
  "docs" : [
    { "_type" : "secilog", "_id" : "1" },
    { "_type" : "secilog", "_id" : "2" }
  ]
}

```

请求：POST http://127.0.0.1:9200/secisland/secilog/_mget?pretty

参数：

```

{
  "docs" : [
    { "_id" : "1" },
    { "_id" : "2" }
  ]
}

```

对于上一种，可以用更加简化的方式查询：

请求：POST http://127.0.0.1:9200/secisland/secilog/_mget?pretty

参数：

```

{"ids" : ["1", "2"]}

```

从上面的例子可以看出，Elasticsearch 的多文档查询还是很灵活的。

2. type 参数说明

在多文档查询中，`_type` 允许为空，它设置为空或者 `_all` 的时候，系统会匹配第一个查询到的结果。如果不设置 `_type`，当有许多文件有相同的 `_id` 的时候，系统最终得到的只有第一个匹配的文档。例如：

请求：POST http://127.0.0.1:9200/secisland/_mget?pretty

参数：

```
{ "ids" : ["1","2" ]}
```

上面的查询当有多个 `type` 中都有 1,2 两个 `id` 的时候，系统只会返回第一个找到的文档。如果想要多个，就需要把 `type` 在请求参数中指出来。

默认情况下，`_source` 字段将在每个文件中返回（如果存储）。类似单个文档的查询，可以在 URL 中指定 `_source`，`_source_include` 或者 `_source_exclude` 来对查询的结果进行过滤。例如：

请求：POST http://127.0.0.1:9200/secisland/secilog/_mget?pretty

参数：

```
{
  "docs" : [
    { "_id" : "1", "_source" : false },
    { "_id" : "2", "_source" : ["collect_type", "collect_date"] }
  ]
}
```

返回值：

```
{
  "docs" : [ {
    "_index" : "secisland",
    "_type" : "secilog",
    "_id" : "1",
    "_version" : 3,
    "found" : true
  }, {
    "_index" : "secisland",
    "_type" : "secilog",
    "_id" : "2",
    "_version" : 1,
    "found" : true,
    "_source" : { "collect_date" : "2016-01-12T09:32:12", "collect_type" :
    "syslog" }
  } ]
}
```

类似单个文档查询，在请求的 URL 中或者参数的 `docs` 中可以指定 `field`、`routing` 参数。

3. 块操作

块操作可以在一个接口中处理文档的内容，包括创建文档、删除文档和修改文档。用块操作方式操作多个文档可以提高系统的效率。例如：

请求：POST http://127.0.0.1:9200/_bulk?pretty

参数：

```
{ "index" : { "_index" : "secisland", "_type" : "secilog", "_id" : "10" } }
{ "field1" : "value1" }
{ "index" : { "_index" : "secisland", "_type" : "secilog", "_id" : "13" } }
{ "field1" : "value3" }
{ "delete" : { "_index" : "secisland", "_type" : "secilog", "_id" : "12" } }
```

返回值：

```
{
  "took" : 1,
  "errors" : false,
  "items" : [ {
    "index" : {
      "_index" : "secisland",
      "_type" : "secilog",
      "_id" : "10",
      "_version" : 6,
      "_shards" : { "total" : 2, "successful" : 1, "failed" : 0 },
      "status" : 200
    }
  }, {
    "index" : {
      "_index" : "secisland",
      "_type" : "secilog",
      "_id" : "13",
      "_version" : 1,
      "_shards" : { "total" : 2, "successful" : 1, "failed" : 0 },
      "status" : 201
    }
  }, {
    "delete" : {
      "_index" : "secisland",
      "_type" : "secilog",
      "_id" : "12",
      "_version" : 2,
      "_shards" : { "total" : 2, "successful" : 1, "failed" : 0 },
      "status" : 404,
      "found" : false
    }
  } ]
}
```

和批量查询类似, `/_bulk`、`{index}/_bulk`、`{index}/{type}/_bulk` 这三种方式都可以执行, 只需要在请求的参数中做出相应的对应。

2.7.5 索引词频率

`term vector` 是在 `Lucene` 中的一个概念, 就是对于文档的某一列, 如 `title`、`body` 这种文本类型的建立词频的多维向量空间, 每一个词就是一个维度, 这个维度的值就是这个词在这个列中的频率。在 `Elasticsearch` 中 `termvectors` 返回在索引中特定文档字段的统计信息, `termvectors` 在 `Elasticsearch` 中是实时分析的, 如果要想不实时分析, 可以设置 `realtime` 参数为 `false`。默认情况下索引词频率统计是关闭的, 需要在建索引的时候手工打开。



注意 在 `Elasticsearch 2.0` 版本以上用 `_termvectors` 代替 `_termvector`。

下面我们建一个打开了索引词统计的索引。

请求: `PUT http://127.0.0.1:9200/secisland/`

参数:

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "type": {
          "type": "string",
          "term_vector": "with_positions_offsets_payloads",
          "store": true,
          "analyzer": "fulltext_analyzer"
        },
        "message": {
          "type": "string",
          "term_vector": "with_positions_offsets_payloads",
          "analyzer": "fulltext_analyzer"
        }
      }
    }
  },
  "settings": {
    "index": { "number_of_shards": 1, "number_of_replicas": 0 },
    "analysis": {
      "analyzer": {
        "fulltext_analyzer": {
          "type": "custom",
          "tokenizer": "whitespace",
          "filter": ["lowercase", "type_as_payload"]
        }
      }
    }
  }
}
```

```

    }
  }
}

```

然后我们插入两条数据。

请求: PUT <http://127.0.0.1:9200/secisland/secilog/1/?pretty>

参数:

```

{
  "type" : "syslog", "message" : "secilog test test test "
}

```

请求: PUT <http://127.0.0.1:9200/secisland/secilog/2/?pretty>

参数:

```

{
  "type" : "file", "message" : "Another secilog test "
}

```

当创建两条日志成功后,我们用 `_termvectors` 来查询统计结果。

请求: GET http://127.0.0.1:9200/secisland/secilog/1/_termvectors?pretty=true

返回值:

```

{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "took" : 2,
  "term_vectors" : {
    "message" : {
      "field_statistics" : {
        "sum_doc_freq" : 5,
        "doc_count" : 2,
        "sum_ttf" : 7
      },
      "terms" : {
        "secilog" : {
          "term_freq" : 1,
          "tokens" : [ {
            "position" : 0,
            "start_offset" : 0,
            "end_offset" : 7,
            "payload" : "d29yZA=="
          } ]
        }
      }
    },
    "test" : {
      "term_freq" : 3,
      "tokens" : [ {

```

}

不是完全

参麦

{

{

```

"fields" : ["message"],
"offsets" : true,
"payloads" : true,
"positions" : true,
"term_statistics" : true,
"field_statistics" : true
}

```

返回值:

```

{
  "_index" : "secisland",
  "_type" : "secilog",
  "_id" : "1",
  "_version" : 1,
  "found" : true,
  "took" : 2,
  "term_vectors" : {
    "message" : {
      "field_statistics" : {"sum_doc_freq" : 5, "doc_count" : 2, "sum_ttf" : 7},
      "terms" : {
        "secilog" : {
          "doc_freq" : 2,
          "ttf" : 2,
          "term_freq" : 1,
          "tokens" : [ {"position" : 0, "start_offset" : 0, "end_offset" : 7,
            "payload" : "d29yZA==" } ]
        },
        "test" : {
          "doc_freq" : 2,
          "ttf" : 4,
          "term_freq" : 3,
          "tokens" : [ {"position" : 1, "start_offset" : 8, "end_offset" : 12,
            "payload" : "d29yZA==" }, {"position" : 2, "start_offset" : 13, "end_offset" : 17,
            "payload" : "d29yZA==" }, {"position" : 3, "start_offset" : 18, "end_offset" : 22,
            "payload" : "d29yZA==" } ]
        }
      }
    }
  }
}

```

从上面的查询中可以看出,对统计进行了过滤,只查询了一部分的统计。

需要注意的是打开了索引词频率会增加系统的负担,除非特别有必要才需要打开统计。

2.7.6 查询更新接口

查询更新接口是 2.3.0 新增的接口,这个接口目前是实验性的。此接口可能会在未来的

版本中改变。此接口在索引中更新每一个文档，而文档的内容并没有改变的情况下使用，这在增加新的属性或者修改映射的时候非常有用。例如：

请求：POST http://127.0.0.0:9200/secisland/_update_by_query?conflicts=proceed

返回值：

```
{
  "took" : 639,
  "updated": 1235,
  "batches": 13,
  "version_conflicts": 2,
  "failures" : [ ]
}
```

`_update_by_query` 当系统开始或者正在用内部版本号进行索引的时候获得一个索引的快照。当快照被处理时或者当索引请求被处理时，如果文档发生了变化，系统将会产生版本冲突。当匹配版本的文档更新后，文档的版本号会递增。当更新和查询失败时，`_update_by_query` 将会中止并在返回值中返回失败的原因。已执行的更新将会保持。也就是说，这个过程是不回滚的，只有中止。当第一条失败时，会导致程序中中止，这意味着所有的更新失败，这时候会返回大量的失败元素。

如果你想当版本冲突时只进行简单的计数，而不中止，可以在 URL 中设置 `conflicts=proceed` 或者在请求内容中设置 `"conflicts": "proceed"`，在 API 中，可以只针对索引中的一个类型进行操作。例如：

请求：POST http://127.0.0.0:9200/secisland/secilog/_update_by_query?conflicts=proceed

同样可以采用 DSL 语法进行查询，例如：

请求：POST http://127.0.0.0:9200/secisland/_update_by_query?conflicts=proceed

```
{
  "query": {
    "term": { "user": "kimchy" }
  }
}
```

这里的搜索和其他搜索的语法是一致的。

之前介绍的都是没有改变文档内容的，其实 `_update_by_query` 是可以支持脚本对文档内容的更新。例如：

请求：POST http://127.0.0.0:9200/secisland/_update_by_query

```
{
  "script": { "inline": "ctx._source.likes++" },
  "query": {
    "term": { "user": "kimchy" }
  }
}
```

这个接口可以在多个索引和多个类型中同时操作，例如：

请求：POST http://127.0.0.0:9200/secisland,blog/secilog,post/_update_by_query

同时，也可以指定路由查询，例如：

请求：POST http://127.0.0.0:9200/secisland/_update_by_query?routing=1

默认情况下，`_update_by_query` 采用滚动 100 批次。你可以在 URL 参数用 `scroll_size` 来改变批次的大小，例如：

请求：POST http://127.0.0.0:9200/secisland/_update_by_query?scroll_size=1000

除了标注的参数外，`_update_by_query` 还支持 `refresh`、`wait_for_completion`、`consistency`、`timeout`。

刷新 (`refresh`) 操作只是当请求完成时再更新所有分片。这不同于更新索引时的刷新，索引的时候当收到新的数据时，只针对当前数据分片进行刷新。如果请求中包含 `wait_for_completion=true`，Elasticsearch 将进行执行前检查，启动请求，然后返回任务，用这个任务可以取消操作或获得任务的状态。一旦请求完成任务就结束了，唯一有记录的地方是在 Elasticsearch 日志文件中有任务的执行结果，这个问题将在以后的版本修复。一致性 (`consistency`) 控制每次请求时有多少分片的拷贝被响应，超时控制每次请求等待的时间。在 Bulk API 中可以精确的知道他们是如何工作的。超时控制多久每批等待成为目标碎片。

每次响应的内容大概是：

```
{
  "took" : 639,
  "updated": 0,
  "batches": 1,
  "version_conflicts": 2,
  "failures" : [ ]
}
```

参数说明如下：

- `took`：从开始到结束的整个操作的毫秒数。
- `updated`：已成功更新的文档数。
- `batches`：滚动响应的数量。
- `version_conflicts`：通过查询命中更新的版本冲突的数量。
- `failures`：所有索引失败的数组。如果这是非空的，则请求中止。

当查询更新操作发生后，可以使用任务接口来获取它们的状态，例如：

请求：POST http://127.0.0.0:9200/_tasks/?pretty&detailed=true&action=*byquery

返回值：

```
{
  "nodes" : {
    "r1A2WoRbTwKZ516z6NEs5A" : {
      "name" : "Tyrannus",
```

```

"transport_address" : "127.0.0.1:9300",
"host" : "127.0.0.1",
"ip" : "127.0.0.1:9300",
"attributes" : {"testattr" : "test", "portsfile" : "true"},
"tasks" : {
  "r1A2WoRbTwKZ516z6NEs5A:36619" : {
    "node" : "r1A2WoRbTwKZ516z6NEs5A",
    "id" : 36619,
    "type" : "transport",
    "action" : "indices:data/write/update/byquery",
    "status" : {
      "total" : 6154,
      "updated" : 3500,
      "created" : 0,
      "deleted" : 0,
      "batches" : 36,
      "version_conflicts" : 0,
      "noops" : 0
    },
    "description" : ""
  }
}
}
}
}

```

该对象包含实际状态。里面有个重要的参数是 **total**，预计重建执行总的操作数。你可以通过添加更新、创建和删除字段来估计这个进展。当它们的总和等于总的列时，该请求将完成。

当创建了一个没有动态映射的索引，如果有新的数据内容，系统会添加新的映射值来匹配数据中的多个字段，例如：

请求：PUT secisland，数据结构只有 secilog

```

{
  "mappings": {
    "secilog": {
      "dynamic": false,
      "properties": {"text": {"type": "string"}}
    }
  }
}

```

插入数据：

请求：POST http://127.0.0.0:9200/secisland/secilog?refresh

```

{"text": "words words", "flag": "bar"}

```

然后得到数据结构：

请求：PUT http://127.0.0.0:9200/secisland/_mapping/secilog

```
{
  "properties": { "text": { "type": "string" },
    "flag": { "type": "string", "analyzer": "keyword" }
  }
}
```

这个时候会自动添加一个 flag 映射。

但这个时候查询数据，不会找到任何值：

请求：POST http://127.0.0.0:9200/secisland/_search?filter_path=hits.total

参数：

```
{ "query": { "match": { "flag": "bar" } } }
```

返回值：

```
{ "hits" : { "total" : 0 } }
```

这个时候我们可以用更新的查询请求来获得数据，例如：

先执行请求：POST http://127.0.0.0:9200/secisland/_update_by_query?refresh&conflicts=proceed

然后执行请求：POST http://127.0.0.0:9200/secisland/_search?filter_path=hits.total

参数：

```
{ "query": { "match": { "flag": "foo" } } }
```

返回值：

```
{ "hits" : { "total" : 1 } }
```

2.8 小结

本章介绍了 Elasticsearch 的索引相关的知识，包括索引管理、索引映射管理等索引高级用法。索引别名管理、类似数据库的索引可以管理多个索引。索引监控、索引状态管理；最后介绍了文档的详细操作，通过以上介绍基本上可以对 Elasticsearch 进行基本的使用。

下一章节重点介绍 Elasticsearch 的映射，通过映射可以了解 Elasticsearch 内部结构。

Chapter 3

第3章

映射

映射是定义存储和索引的文档类型以及字段的过程。索引中的每一个文档都有一个类型，每种类型都有它自己的映射。一个映射定义了文档结构内每个字段的数据类型。映射通过配置来定义字段类型与该类型相关联的元数据的关系。例如，可以通过映射来定义日期类型的格式、数字类型的格式或者文档中所有字段的值是否应该被 `_all` 字段索引等。本章将介绍映射的概念、参数，以及动态映射的使用等。

3.1 概念

1. 映射类型

每个索引拥有一个或多个映射类型，用来在索引中将文档划分为不同的逻辑组。

每个映射类型拥有：

- 元字段：用来定义如何处理文档的元数据。元字段包括文档的 `_index` 字段、`_type` 字段、`_id` 字段和 `_source` 字段。
- 字段或属性：每个映射类型包含与类型相关的字段或属性列表。同一索引中不同映射类型的相同名称字段必须拥有相同的映射。

2. 字段数据类型

每个字段拥有一个数据类型，可以是简单数据类型，比如字符串型（`String`）、日期型（`date`）、长整型（`long`）、双精度浮点型（`double`）、布尔型（`boolean`）或者 IP。

支持 JSON 的层次性类型，比如对象（`object`）、嵌套（`nested`），或者特定的类型，比如地理点（`geo_point`）、地理形状（`geo_shape`）。

基于不同目的对同一个字段进行不同方式的索引是很有用的。例如，一个字符串类型字段可以在全文搜索中作为分析字段，在排序或聚合时作为不分析的字段。或者，可以通过标准分析器、英文分析器或者法语分析器对字符串字段进行索引。

一个数据类型通过 `fields` 参数支持多字段。

3. 动态映射

字段和映射类型在使用前不需要事先定义。依靠动态映射，通过索引文档，新的映射类型和字段名会自动添加。新的字段可以添加到顶级映射类型或者映射内部的对象和嵌入字段。

动态映射可以配置自定义映射用于新类型或者新字段。

4. 显式映射

相对于 Elasticsearch 来说，我们对于数据类型的掌控更加全面，所以我们可以指定显式映射而不是使用动态映射。

当创建索引的时候，可以创建映射类型和字段。也可以在当前的索引中通过映射创建接口添加映射类型和字段。

5. 更新当前映射

除了记录之外，现有的映射类型和字段不能更新。修改映射意味着废弃已经索引的文档，我们反而应该根据映射创建新的索引并且重新索引数据。

6. 映射类型之间共享字段

映射类型在每个索引中是唯一的，就是在一个索引的多个类型中，如果多个类型中的映射名称一样，则它必须是相同的类型。

例如：如果一个 `title` 字段同时存在于 `user` 和 `blogpost` 映射类型中，`title` 字段在每个类型中必须拥有相同的映射。这个规则的唯一例外是：对于 `copy_to` 参数、`dynamic` 参数、`enabled` 参数、`ignore_above` 参数，`include_in_all` 参数，每个不同映射类型中的字段拥有不同的参数设置。

通常，相同名称的字段由相同类型的数据构成，所以拥有相同的索引是没有问题的。当产生类型冲突的时候，可以选择更详细的命名，比如 `user_title` 和 `blog_title`。

7. 映射示例

当创建索引的时候，可以指定映射：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "user": {
      "_all": { "enabled": false },
      "properties": {
        "title": { "type": "string" },

```

```

    "name": { "type": "string" },
    "age": { "type": "integer" }
  },
  "blogpost": {
    "properties": {
      "title": { "type": "string" },
      "body": { "type": "string" },
      "user_id": { "type": "string", "index": "not_analyzed" },
      "created": {
        "type": "date",
        "format": "strict_date_optional_time||epoch_millis"
      }
    }
  }
}

```

上面的接口表示创建一个名为 `secisland` 的索引，在索引中添加名为 `user` 和 `blogpost` 的映射类型。`user` 映射类型取消元字段 `_all`，指定了每个映射类型的字段或属性，指定了每个字段的数据类型和映射。

3.2 字段数据类型

Elasticsearch 支持一系列不同的数据类型来定义文档字段，分为核心数据、复杂数据、地理数据、专门数据类型。

核心数据类型包括：

- ❑ 字符串数据类型：string
- ❑ 数字型数据类型：long、integer、short、byte、double、float
- ❑ 日期型数据类型：date
- ❑ 布尔型数据类型：boolean
- ❑ 二进制数据类型：binary

复杂数据类型包括：

- ❑ 数组数据类型：不需要专门的类型来定义数组。
- ❑ 对象数据类型：object，单独的 JSON 对象。
- ❑ 嵌套数据类型：nested，关于 JSON 对象的数组。

地理数据类型包括：

- ❑ 地理点数据类型：geo_point，经纬点。
- ❑ 地理形状数据类型：geo_shape，多边形的复杂地理形状。

专门数据类型包括：

- ❑ IPv4 数据类型：IP 协议为 IPv4 的地址。

- 完成数据类型: `completion`, 提供自动补全的建议。
- 单词计数数据类型: `token_count`, 统计字符串中的单词数量。

3.2.1 核心数据类型

1. 字符串数据类型

字符串数据类型的字段接受文本值, 可以分为如下两种:

- **全文本**。全文本值通常用于基于文本的相关性搜索, 全文本字段可以分词, 即在索引执行之前通过一个分词器将字符串转换为单词列表。分词操作使得 Elasticsearch 可以在全文本字段上搜索单词。全文本字段不用于排序而且很少用于聚合。
- **关键字**。关键字是个精准值, 通常用于过滤 (例如, 为 `published` 的博客文章获取所有 `status` 字段值)、排序、参与聚合。关键字字段不参与分词。

全文本 (可以分词) 字段和关键字 (不可以分词) 字段映射示例如下:

请求: `PUT http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "full_name": { "type": "string" },
        "status": { "type": "string", "index": "not_analyzed" }
      }
    }
  }
}
```

其中, `full_name` 字段是一个可分词的全文本类型字段——`index`: 默认是 `analyzed`。
`status` 字段是一个不可分词的关键字字段。

同一个字段同时拥有全文本和关键字两个版本, 通常是很有用的: 一个用于全文搜索, 另一个用于聚合和排序。这可以通过多字段来实现。

字符串数据类型的字段可以接受的参数如表 3-1 所示。

表 3-1 字符串数据类型的字段可以接受的参数

参数	说明
<code>analyzer</code>	分词器可以用于可分词的字符串型字段。默认为默认的索引分词器或者标准分词器
<code>boost</code>	字段级索引加权。接受浮点型数字, 默认值是 1.0
<code>doc_values</code>	定义字段是否应该以列跨度的方式存储在磁盘上, 以便用于排序、聚合或者脚本。接受 <code>true</code> 或 <code>false</code> 参数。对于不可分词字段, 默认值是 <code>true</code> 。可分词字段不支持这个参数
<code>fieldtype</code>	决定字段是否可以使用内存字段值进行排序, 聚合或者在脚本中使用。接受 <code>disabled</code> 或者 <code>paged_bytes</code> (默认) 参数。没有分析过的字段会优先使用文档值

(续)

参数	说明
ignore_above	不要索引或执行任何长于这个值的字符串。默认为 0 (禁用)
include_in_all	决定字段是否应该包含在 _all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false, 参数默认值为 false; 其他情况下, 默认值为 true
index	决定字段是否可以被用户搜索。接受参数 analyzed (默认, 视为全文本字段), not_analyzed (作为关键字字段) 以及 no
index_options	定义存储在索引中, 用于搜索和突出用途的信息
norms	计算查询得分的时候是否应该考虑字段长度。默认依赖于索引设置: analyzed 字段默认为 { "enabled": true, "loading": "lazy" }。not_analyzed 字段默认为 { "enabled": false }
null_value	接受一个字符串值替换所有 null 值。默认为 null, 意味着字段作为缺失字段。如果字段是可分词 (analyzed) 的, null_value 也会被分词
position_increment_gap	定义字符串数组中应该插入的虚拟索引词的数量。默认值为 100, 以一个较合理的值来阻止短语查询在跨字段匹配索引词的时候溢出
store	决定字段值是否应该被存储以及从 _source 字段分别获取。接受参数 true 或 false (默认)
search_analyzer	指定搜索时用在可分词字段上的分词器
search_quote_analyzer	指定搜索短语时使用的分词器
similarity	指定使用的相似度评分算法, 默认为 TF/IDF
term_vector	定义一个可分词字段是否应该存储索引词向量。默认为 no

2. 数字型数据类型

数字型数据类型支持的数字类型如表 3-2 所示。

表 3-2 数字型数据类型

参数	说明
long	一个有符号的 64 位整数, 最小值为 -2^{63} , 最大值为 $2^{63}-1$
integer	一个有符号的 32 位整数, 最小值为 -2^{31} , 最大值为 $2^{31}-1$
short	一个有符号的 16 位整数, 最小值为 -32 768, 最大值为 32 767
byte	一个有符号的 8 位整数, 最小值为 -128, 最大值为 127
double	64 位双精度浮点数
float	32 位单精度浮点数

数字型字段映射配置示例如下:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "number_of_bytes": {"type": "integer"},

```



```

    "time_in_seconds": {"type": "float"}
  }
}
}

```

数字型字段参数见表 3-3。

表 3-3 数字型字段参数

参数	说明
coerce	试着将字符串型数据转换为整数型数字数据
boost	字段级索引加权, 接受浮点型数字参数, 默认为 1.0
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上, 以便用于排序、聚合或者脚本。接受 true (默认) 或 false 参数
ignore_malformed	如果是 true, 畸形的数字会被忽略。如果是 false (默认), 畸形数字会抛出异常并丢弃整个文档
include_in_all	决定字段是否应该包含在 _all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false, 参数默认值为 false; 其他情况下, 默认值为 true
index	决定字段是否可以被用户搜索。接受参数 not_analyzed (默认) 以及 no
null_value	接受与字段同类型的数字型值来代替 null 值。默认是 null, 意味着字段作为缺失字段
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值取决于数字类型
store	决定字段值是否应该存储以及从 _source 字段分别获取。接受参数 true 或 false (默认)

3. 日期型数据类型

JSON 没有日期型数据类型, 所以在 Elasticsearch 中, 日期可以是:

- 包含格式化日期的字符串, 例如“2015-01-01”或者“2015/01/01 12:10:30”。
- 代表时间毫秒数的长整型数字。
- 代表时间秒数的整数。

通常, 日期被转换为 UTC (如果时区被指定) 但是存储为代表时间毫秒数的长整数。

可以自定义时间格式, 如果没有指定格式, 则使用默认值:

```
"strict_date_optional_time||epoch_millis"
```

这意味着接受任意时间戳的日期值, 例如:

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "date": {"type": "date"}
      }
    }
  }
}

```


创建映射之后，可以放置日期型数据：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "date": "2015-01-01" }
```

请求：PUT http://127.0.0.1:9200/secisland/secilog/2

```
{ "date": "2015-01-01T12:10:30Z" }
```

请求：PUT http://127.0.0.1:9200/secisland/secilog/3

```
{ "date": 1420070400001 }
```

(1) 多日期格式

使用双竖线(∥)分隔，可以指定多个日期格式。每个格式会被依次尝试，直到找到匹配的格式。第一个格式会用于将时间毫秒数值转换为字符串：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "date": { "type": "date",
                  "format": "yyy-MM-dd HH:mm:ss|yyyy-MM-dd|epoch_millis"
                }
      }
    }
  }
}
```

(2) 日期型数字的字段参数

日期型数据的字段参数参见表 3-4。

表 3-4 日期型字段参数

参数	说明
boost	字段级索引加权，接受浮点型数字参数，默认为 1.0
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本。接受 true（默认）或 false 参数
format	可解析的日期格式。默认为 strict_date_optional_time epoch_millis
ignore_malformed	如果是 true，畸形的日期会被忽略。如果是 false（默认），畸形日期会抛出异常并丢弃整个文档
include_in_all	决定字段是否应该包含在 _all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no
null_value	接受日期型值来代替 null 值。默认是 null，意味着字段作为缺失字段
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值为 16
store	决定字段值是否应该存储以及从 _source 字段分别获取。接受参数 true 或 false（默认）

4. 布尔数据类型

布尔型字段接受 true 或 false 值，也可以接受代表真或假的字符串和数字：

□ 假值——false, “false”, “off”, “no”, “0”, “”(空字符串), 0, 0.0。

□ 真值——其他任何非假的值。

示例如下：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "is_published": {"type": "boolean"}
      }
    }
  }
}
```

请求：POST http://127.0.0.1:9200/secisland/secilog/1

```
{"is_published": true }
```

索引词聚合之类的聚合使用 1 和 0 作为 key，使用字符串 “true” 和 “false” 作为 key_as_string。使用脚本时，布尔字段返回 1 和 0。布尔型字段参数见表 3-5。

表 3-5 布尔型字段参数

参数	说明
boost	字段级索引加权，接受浮点型数字参数，默认为 1.0
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序、聚合或者脚本。接受 true (默认) 或 false 参数
index	决定字段是否可以被用户搜索。接受参数 not_analyzed (默认) 以及 no
null_value	接受布尔型值来代替 null 值。默认是 null，意味着字段被作为缺失字段
store	决定字段值是否应该被存储以及从 _source 字段分别获取。接受参数 true 或 false (默认)

5. 二进制数据类型

二进制数据类型接受 Base64 编码字符串的二进制值。字段不以默认方式存储而且不能搜索：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "name": {"type": "string"},
        "blob": {"type": "binary"}
      }
    }
  }
}
```

```

    }
  }
}
请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```

```

{
  "name": "Some binary blob",
  "blob": "U29tZSBiaW5hcnkgYmxvYg=="
}

```

Base64 编码二进制值不能嵌入换行符 `\n`。二进制数据类型的字段参数如下所示:


- ❑ `doc_values`——定义字段是否应该以列跨度的方式存储在磁盘上,以便用于排序、聚合或者脚本。接受 `true` (默认) 或 `false` 参数。
- ❑ `store`——决定字段值是否应该存储以及从 `_source` 字段分别获取。接受参数 `true` 或 `false` (默认)。

3.2.2 复杂数据类型

1. 数组数据类型

在 Elasticsearch 中,没有专门的数组类型。每个字段默认可以包含零个或更多的值,然而,数组中所有的值都必须是相同的数据类型。例如:

- ❑ 字符串数组: `["one", "two"]`
- ❑ 整数数组: `[1,2]`
- ❑ 由数组组成的数组: `[1,[2,3]]`, 等同于 `[1,2,3]`
- ❑ 对象数组: `[{ "name": "Mary", "age": 12 }, { "name": "John", "age": 10 }]`

 **注意** 无法对数组中的每一个对象进行单独的查询。

当动态添加字段的时候,数组中第一个元素的值决定了字段类型,随后的所有值必须是相同的数据类型或者可以强制转换为相同的数据类型。

Elasticsearch 不支持混合数据类型的数组,比如: `[10, "some string"]`。

数组可能包含 `null` 值,会被 `null_value` 配置替换掉或者忽略掉。一个空数组 `[]` 被当作缺失字段——没有值的字段。

2. 对象数据类型

JSON 文档是天然分层的:文档可以包含内部对象。同样,内部对象也可以包含内部对象。

请求: PUT `http://127.0.0.1:9200/secisland/secilog/1`

```

{

```

```

"region": "US",
"manager": {
  "age": 30,
  "name": { "first": "John", "last": "Smith" }
}

```

本质上，文档被简单地索引为键值对的列表，形如：

```

{
  "region": "US",
  "manager.age": 30,
  "manager.name.first": "John",
  "manager.name.last": "Smith"
}

```

上面文档的映射结构为：

请求：PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "region": {
          "type": "string",
          "index": "not_analyzed"
        },
        "manager": {
          "properties": {
            "age": { "type": "integer" },
            "name": {
              "properties": { "first": { "type": "string" },
                             "last": { "type": "string" }
              }
            }
          }
        }
      }
    }
  }
}

```

映射类型是一种对象类型，拥有参数字段，`manager` 字段是一个内部对象字段，`manager.name` 字段是 `manager` 字段中的内部对象字段。可以明确地设置 `type` 字段为 `object` (默认值)。对象数据类型的参数如下所示：

- ☐ `dynamic`——定义新的参数是否应该动态加入到已经存在的对象中。接受 `true` (默认)，`false` 和 `strict`。
- ☐ `enabled`——赋值给对象字段的 JSON 值应该被解析和索引 (`true`，默认) 还是完全忽略 (`false`)。
- ☐ `include_in_all`——为对象内的所有属性设置 `include_in_all` 值。对象本身不添加到 `_all` 字段。

□ **properties**——对象内的字段可以是任意数据类型，包括对象数据类型。新的属性可以添加到已存在的对象中。

3. 嵌套数据类型

嵌套数据类型是对象数据类型一个专门的版本，用来使一组对象被单独地索引和查询。

(1) 对象数组是如何摊平的

Lucene 没有内部对象的概念，所以 Elasticsearch 利用简单的列表存储字段名和值，将对象层次摊平。例如，下面的文档：

请求：PUT <http://127.0.0.1:9200/secisland/secilog/1>

```
{
  "group" : "fans",
  "user" : [
    { "first" : "John", "last" : "Smith" },
    { "first" : "Alice", "last" : "White" }
  ]
}
```

把它内部转换为文档，结构如下：

```
{
  "group" : "fans",
  "user.first" : [ "alice", "john" ],
  "user.last" : [ "smith", "white" ]
}
```

user.first 和 user.last 字段存在多值字段中，alice 和 white 的关联性丢失了。这个文档可能错误地匹配到关于 alice 和 smith 的查询。

(2) 对一组对象使用嵌套字段

如果需要对一组对象进行索引而且保留数组中每个对象的独立性，可以使用嵌套数据类型而不是对象数据类型。本质上，嵌套对象将数组中的每个对象作为分离出来的隐藏文档进行索引。这也意味着每个嵌套对象可以独立于其他对象被查询，示例如下。

创建的映射如下所示：

请求：PUT <http://127.0.0.1:9200/secisland>

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "user": { "type": "nested" }
      }
    }
  }
}
```

插入的数据如下所示：

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{
  "group": "fans",
  "user": [
    { "first": "John", "last": "Smith" },
    { "first": "Alice", "last": "White" }
  ]
}
```

可以根据对象进行搜索,但对象的条件要全匹配才能搜到,例如搜索 1 如下所示:

请求: POST http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "nested": {
      "path": "user",
      "query": {
        "bool": {
          "must": [
            { "match": { "user.first": "Alice" } },
            { "match": { "user.last": "Smith" } }
          ]
        }
      }
    }
  }
}
```

搜索 2 如下所示:

请求: POST http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "nested": {
      "path": "user",
      "query": {
        "bool": {
          "must": [
            { "match": { "user.first": "Alice" } },
            { "match": { "user.last": "White" } }
          ]
        }
      }
    }
  }
}
```

user 字段作为嵌套类型添加到索引中。搜索 1 匹配不到结果,因为 Alice 和 Smith 不在同一个嵌套对象中;搜索 2 匹配到搜索结果,因为 Alice 和 White 在同一个嵌套对象中。

嵌套数据类型的字段参数如下所示:

- ❑ **dynamic**——定义新的参数是否应该动态加入到已经存在的对象中。接受 **true** (默认), **false** 和 **strict**。
- ❑ **include_in_all**——设置所有嵌套对象属性的 **include_in_all** 值。嵌套文档没有它们自身的 **_all** 字段, 取而代之的是, 值被添加到“根”文档的 **_all** 字段中。
- ❑ **properties**——嵌套对象的字段可以是任何数据类型, 包括嵌套对象类型。新的属性可以被添加到已经存在的嵌套对象中。

3.2.3 地理数据类型

1. 地理点数据类型

地理点数据类型字段接受经纬度对, 可用于:

- ❑ 查找一定范围内的地理点, 这个范围可以是相对于一个中心点的固定距离, 也可以是多边形或者地理散列单元。
- ❑ 通过地理位置或者相对于中心点的距离聚合文档。
- ❑ 整合距离到文档的相关性评分中。
- ❑ 通过距离对文档进行排序。

指定字段类型为地理位置数据类型:

请求: **PUT http://127.0.0.1:9200/secisland**

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "location": { "type": "geo_point" }
      }
    }
  }
}
```

存储地理位置数据有 4 种不同方式。下面分别介绍。

(1) 请求: **PUT http://127.0.0.1:9200/secisland/secilog/1**

```
{"text": "Geo-point as an object", "location": { "lat": 41.12, "lon": -71.34 }}
```

地理点参数形如对象参数, 拥有纬度和经度键值对。

(2) 请求: **PUT http://127.0.0.1:9200/secisland/secilog/2**

```
{"text": "Geo-point as a string", "location": "41.12, -71.34" }
```

字符串地理点参数的格式为“纬度, 经度”。

(3) 请求 **PUT http://127.0.0.1:9200/secisland/secilog/3**

```
{"text": "Geo-point as a geohash", "location": "drm3btev3e86" }
```

散列地理点参数。

(4) 请求: PUT http://127.0.0.1:9200/secisland/secilog/4

```
{ "text": "Geo-point as an array", "location": [ -71.34, 41.12 ] }
```

地理点数组参数, 格式为 [经度, 纬度]。


 **注意** 字符串地理点参数顺序为 (纬度, 经度), 地理点数组参数的顺序为 (经度, 纬度)。地理点数据字段参数见表 3-6。

表 3-6 地理点字段参数

参数	说明
coerce	基于标准的 -180:180/-90:90 坐标系统的经度和纬度值。接受 true 和 false (默认)
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上, 以便用于排序、聚合或者脚本。接受 true (默认) 或 false 参数
geohash	定义地理点是否应该作为地理散列值在子字段 .geohash 中被索引。默认值为 false, 除非 geohash_prefix 参数值为 true
geohash_precision	用于 geohash 和 geohash_prefix 选项的地理散列最大长度
geohash_prefix	定义地理点是否应该作为添加前缀的地理散列来进行索引。默认值为 false
ignore_malformed	如果是 true, 畸形的地理点会被忽略。如果是 false (默认), 畸形地理点会抛出异常并丢弃整个文档
lat_lon	定义地理点是否应该在子字段 .lat 和 .lon 中被索引到。接受 true 和 false (默认)
precision_step	控制每个经纬点被索引的额外索引词的数量。默认值为 16。与 lat_lon 参数的值无关

2. 地理形状数据类型

地理形状数据类型有利于索引和搜索任意地理形状, 例如矩形和多边形。无论是数据被索引还是在查询执行的过程中, 都可以使用地理形状数据类型在地理点的基础上包含地理形状。

利用地理形状查询 (geo_shape) 来查询文档, 可以使用地理形状数据类型。

(1) 映射选项

地理形状类型映射将 geo_json 几何对象映射成地理形状类型。为了启用映射选项, 需要明确映射字段为地理形状类型, 参见表 3-7。

(2) 前缀树

在索引中高效地表示形状, 形状被转换到一系列表示为方格 (通常被称为“栅格”) 的散列用于实现一个前缀树。树的概念来自该前缀树使用多层网格, 每层增加精度级别来表示陆地。这可以提高地图的缩放级别或图像的细节水平。

(3) 空间策略

前缀树的实现基于空间策略用来分解提供的形状为近似方格。每个策略解决这些问题:

□ 哪种类型的形状可以被索引。

- ❑ 形状可以用于哪种类型的查询操作。
 - ❑ 每个字段是否可以保存多个形状。
- 提供的这些策略实现（具有相应的功能）见表 3-8。

表 3-7 地理形状数据类型映射选项

选项	描述	默认
tree	引用的前缀树名: geohash 或 quadtree	geohash
precision	这个参数可以用来代替 tree_levels 来设置一个适当的 tree_levels 参数值。指定一个适当的精确度, Elasticsearch 会计算匹配精确度的最佳 tree_levels 值。这个值应该是一个数字, 后面跟着一个可选的距离单元。可用的距离单元包括: in、inch、yd、yard、mi、miled、km、kilometers、m、meters、cm、centimeters、mm、millimeters	meters
tree_levels	前缀树使用的层的最大数量。可以用来控制形状表示的精度以及因此索引的索引词数量。默认是选取前缀树引用的默认值。因为这个参数需要一定程度的底层实现的理解, 可以使用 precision 参数进行替代。然而, 即使使用 precision 参数, Elasticsearch 本质上只会使用并通过映射接口返回 tree_levels	50m
strategy	定义了如何在索引和搜索时表示形状。会影响可用的功能, 所以建议让 Elasticsearch 自动设置这个参数。有两种可用的值: recursive 和 term。term 仅支持点类型 (points_only 参数会自动设置为 true), recursive 支持所有的形状类型	recursive
distance_error_pct	用于示意前缀树应该使用的精确值。默认是 0.025 (2.5%), 最大支持的值为 0.5	0.025
orientation	定义了如何解读多边形 / 多边形集合顶点的顺序。这个参数定义两种坐标系统规则 (右手或左手) 中的一个, 每一种坐标系统可以用三种方式进行指定。1. 右手规则: right、ccw、counterclockwise, 2. 左手规则: left、cw、clockwise。默认方向 (counterclockwise) 遵循 OGC 标准, 定义了外环顶点按照逆时针的方向内环顶点按顺时针方向。在映射中对地理形状字段设置这个参数, 可以明确设置坐标列表中的顶点顺序	ccw
points_only	设置这个选项为 true (默认为 false), 只对点形状配置地理形状字段类型 (不支持多个点)	false

表 3-8 空间策略

策略	支持的形状	支持的查询	多形状
recursive	所有	INTERSECTS、DISJOINT、WITHIN、CONTAINS	是
term	点	INTERSECTS	是

(4) 准确性

地理形状不提供 100% 的准确性, 并且取决于匹配值, 可能对确定的查询返回一些误判或漏判的结果。为了缓和这个问题, 需要为 tree_levels 参数选择一个合适的值来适应相应的预期。

例如:

```
{
  "properties": {
```

```

    "location": { "type": "geo_shape", "tree": "quadtree", "precision": "1m" }
  }
}

```

这个映射将 location 字段映射到地理形状类型，使用 quad 前缀树并且精度为 1m。Elasticsearch 转换这个精度到 tree_levels 设置为 26。

(5) 性能方面的考虑

Elasticsearch 使用前缀树中的路径作为索引和查询中的索引词。更高级别的精确值，会生成更多的索引词。计算索引词、加载到内存、保存到磁盘也需要额外的性能开销。

索引大小和合理水平的精确值的折中是 50m。

(6) 输入结构

用于表示形状的 GeoJSON 格式见表 3-9:

表 3-9 GeoJSON 格式

GeoJSON 类型	Elasticsearch 类型	描述
Point	point	单独地理坐标点
LineString	linestring	一个任意的线，给出两个或更多的点
Polygon	polygon	一个封闭的多边形，第一个点和最后一个点必须匹配，需要 N+1 个顶点创建一个 N 多边形，最少需要 4 个顶点
MultiPoint	multipoint	一组不连续的但是可能相关的点
MultiLineString	multilinestring	一组分离的线
MultiPolygon	multipolygon	一组分离的多边形
GeometryCollection	geometrycollection	一种类似于 multi* 形状的 GeoJSON 形状，多种类型可以共存（例如，一个点和一条线）
N/A	envelope	一个封闭的矩形，通过指定左上角和右下角来确定
N/A	circle	一个圆，通过指定圆心和带单位的半径来确定，默认单位为 METERS

下面举例说明这些格式。

Point 是一个单独的地理坐标点，比如当前建筑的位置或者智能手机地理定位接口提供的确切位置。

```

{
  "location": {
    "type": "point",
    "coordinates": [-77.03653, 38.897676]
  }
}

```

Linestring 通过两个或更多的一组位置定义。只指定两个点，Linestring 会表示一条直线。指定更多的点，可以创建任意的线。

```

{
  "location": {

```



```

    "type" : "linestring",
    "coordinates" : [[-77.03653, 38.897676], [-77.009051, 38.889939]]
  }
}

```

Polygon 通过一系列地理点列表进行定义。每个列表（外环）中的第一个点和最后一个点必须相同（多边形必须是封闭的）。

```

{
  "location" : {
    "type" : "polygon",
    "coordinates" : [
      [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ]
    ]
  }
}

```

第一个数组表示多边形的外环边界，其他数组表示内部形状（“孔”）。

```

{
  "location" : {
    "type" : "polygon",
    "coordinates" : [
      [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],
      [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ]
    ]
  }
}

```



注意 GeoJSON 不强制指定顶点的顺序，而在国际日期变更线和极点附近的多边形是极有可能造成混乱的。为了解决这种混乱，开放地理空间信息联盟（OGC）简单特征访问规范定义了这些顶点顺序：

□ 外环：逆时针方向。

□ 内环 / 孔：顺时针方向。

对于不跨越国际日期变更线的多边形，顶点的顺序对于 Elasticsearch 来说是没有关系的。Elasticsearch 完全按照 OGC 制定的规范请求顶点，否则，会创建一个意外的多边形，并且返回不符合预期的查询 / 过滤结果。

orientation 参数可以在地理形状数据类型字段创建映射的时候进行定义，规定顶点的顺序。也可以在每个文档中进行重写：

```

{
  "location" : {
    "type" : "polygon",
    "orientation" : "clockwise",
    "coordinates" : [
      [ [-177.0, 10.0], [176.0, 15.0], [172.0, 0.0], [176.0, -15.0], [-177.0,

```

```

    -10.0], [-177.0, 10.0] ], [ [178.2, 8.2], [-178.8, 8.2], [-180.8, -8.8],
    [178.2, 8.8] ]
  ]
}

```

MultiPoint 是 GeoJSON 点的列表, 如下所示:

```

{
  "location" : { "type" : "multipoint", "coordinates" : [[102.0, 2.0], [103.0, 2.0]] }
}

```

MultiLineString 是 GeoJSON 线的列表, 如下所示:

```

{
  "location" : { "type" : "multilinestring", "coordinates" : [
    [ [102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0] ],
    [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0] ],
    [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8] ] ]
}

```

MultiPolygon 是 GeoJSON 多边形的列表, 如下所示:

```

{
  "location" : {
    "type" : "multipolygon",
    "coordinates" : [
      [ [102.0, 2.0], [103.0, 2.0], [103.0, 3.0], [102.0, 3.0], [102.0, 2.0] ] ],

      [ [ [100.0, 0.0], [101.0, 0.0], [101.0, 1.0], [100.0, 1.0], [100.0, 0.0] ],
        [ [100.2, 0.2], [100.8, 0.2], [100.8, 0.8], [100.2, 0.8], [100.2, 0.2] ] ]
    ]
  }
}

```

GeometryCollection 是 GeoJSON 地理几何对象的集合, 如下所示:

```

{
  "location" : {
    "type": "geometrycollection",
    "geometries": [
      { "type": "point", "coordinates": [100.0, 0.0] },
      { "type": "linestring", "coordinates": [ [101.0, 0.0], [102.0, 1.0] ] }
    ]
  }
}

```

Envelope 包含矩形左上角和右下角的坐标值来表示矩形, 例如:

```

{
  "location" : { "type" : "envelope",
    "coordinates" : [ [-45.0, 45.0], [45.0, -45.0] ] }
}

```

Circle 包含圆心和半径:

```
{
  "location" : {
    "type" : "circle", "coordinates" : [-45.0, 45.0], "radius" : "100m"}
}
```

注意, radius 是必要字段。距离的单位默认为米 (METERS)。

(7) 排序和取回形状索引

由于形状复杂的输入结构和索引表示, 当前不能直接通过字段排序或取回形状。地理形状值只能通过 _source 字段取回。

3.2.4 专门数据类型

1. IPv4 数据类型

IPv4 字段本质上是一个长整型字段, 接受 IPv4 地址并作为长整型值进行索引。

添加映射:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "ip_addr": {"type": "ip"}
      }
    }
  }
}
```

插入数据:

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{"ip_addr": "192.168.1.1"}
```

IPv4 数据类型的字段参数见表 3-10。

表 3-10 IP 字段参数

参数	说明
boost	字段级索引加权, 接受浮点型数字参数, 默认为 1.0
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上, 以便用于排序、聚合或者脚本。接受 true (默认) 或 false 参数
include_in_all	决定字段是否应该包含在 _all 字段中。接受 true 或 false 参数。如果索引设置为 no 或者父对象字段设置 include_in_all 为 false, 参数默认值为 false; 其他情况下, 默认值为 true
index	决定字段是否可以被用户搜索。接受参数 not_analyzed (默认) 以及 no
null_value	接受一个 IPv4 值替换所有 null 值。默认为 null, 意味着字段作为缺失字段
precision_step	控制索引的额外索引词的数量来使范围查询更快。默认值为 16
store	决定字段值是否应该存储以及从 _source 字段分别获取。接受参数 true 或 false (默认)

注: IPv6 地址现在还不提供支持。

2. 单词计数数据类型

单词计数型字段本质上是一个整数字段，接受并分析字符串值，然后索引字符串中单词的个数。下面举例说明。

添加映射：

请求：PUT <http://127.0.0.1:9200/secisland>

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "name": {
          "type": "string",
          "fields": {
            "length": { "type": "token_count", "analyzer": "standard" }
          }
        }
      }
    }
  }
}
```

插入数据：

请求：PUT <http://127.0.0.1:9200/secisland/secilog/1>

```
{ "name": "John Smith" }
```

严格来说，单词计数类型计算位置增量而不是统计单词。这意味着即使分析器过滤掉一部分单词，它们也会被包含在计数中。单词计数类型的字段参数如表 3-11 所示。

表 3-11 单词计数型字段参数

参数	说明
analyzer	必要字段，定义用来分析字符串值的分析器
boost	字段级索引加权，接受浮点型数字参数，默认为 1.0
doc_values	定义字段是否应该以列跨度的方式存储在磁盘上，以便用于排序，聚合或者脚本？接受 true（默认）或 false 参数
include_in_all	决定字段是否应该被包含在 _all 字段中。接受 true 或 false 参数。如果索引被设置为 no 或者父对象字段设置 include_in_all 为 false，参数默认值为 false；其他情况下，默认值为 true
index	决定字段是否可以被用户搜索。接受参数 not_analyzed（默认）以及 no
null_value	接受一个与字段相同类型的数字型值替换所有 null 值。默认为 null，意味着字段被作为缺失字段
precision_step	控制索引的额外索引词的数量来使范围查询更快速。默认值为 32
store	决定字段值是否应该被存储以及从 _source 字段分别获取。接受参数 true 或 false（默认）

请求：PUT <http://127.0.0.1:9200/secisland/secilog/1>

```
{ "title": "This is a test document", "body": "This document has a body!" }
```

请求：GET http://127.0.0.1:9200/secisland/_search

3.3 元字段

每个文档都有与之关联的元数据，元字段是为了保证系统正常运转的内置字段，比如 `_index` 表示索引字段，`_type` 表示映射类型字段和 `_id` 表示文档主键字段，这些字段都是以下划线开头的。当映射类型被创建的时候，可以自定义一些元字段的行为，例如标识元字段、文档来源元字段、索引元字段、路由元字段等。参见表 3-12 到表 3-16。

表 3-12 标识元字段

参数	说明
<code>_index</code>	文档所属的索引
<code>_uid</code>	包含 <code>_type</code> 和 <code>_id</code> 的混合字段
<code>_type</code>	文档的映射类型
<code>_id</code>	文档的 ID

表 3-13 文档来源元字段

参数	说明
<code>_source</code>	作为文档内容的原始 JSON
<code>_size</code>	<code>_source</code> 元字段占用的字节数，通过 <code>mapper-size</code> 插件提供

表 3-14 索引元字段

参数	说明
<code>_all</code>	索引所有字段的值
<code>_field_names</code>	文档中所有包含非空值的字段
<code>_timestamp</code>	关联文章的时间戳，可以手动指定或者自动生成
<code>_ttl</code>	定义文档被自动删除之前的存活时间

表 3-15 路由元字段

参数	说明
<code>_parent</code>	用于在映射类型之间创建父子关系
<code>_routing</code>	一个自定义的路由值，路由由文档到一个特定的分片

表 3-16 其他元字段

参数	说明
<code>_meta</code>	应用特定的元字段

下面介绍一些常用字段。

3.3.1 _all 字段

_all 字段是一个特殊的包含全部内容的字段，在一个大字符串中关联所有其他字段的值，使用空格作为分隔符。可以被分析和索引但不会被存储。使用 _all 字段可以对文档的值进行搜索而不必知道包含所需值的字段名。当面对一个新的数据集的时候，_all 字段是非常有用的选项。

插入数据：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{
  "first_name": "John",
  "last_name": "Smith",
  "date_of_birth": "1970-10-24"
}
```

利用 _all 字段进行搜索：

请求：GET http://127.0.0.1:9200/secisland/_search

```
{"query": {"match": {"_all": "john smith 1970"}}}
```

_all 字段包含的索引词：["john", "smith", "1970", "10", "24"]

注解：date_of_birth 字段作为日期型字段，会索引一个索引词 1970-10-24 00:00:00 UTC。但是，_all 字段将所有的值作为字符串，所以日期值作为三个字符串被索引："1970", "10", "24"。

_all 字段就是一个字符串类型字段，接受与字符串型字段相同的参数，包括 analyzer, index_options 和 store。

_all 字段关联字段值的时候，丢失了短字段（高相关性）和长字段（低相关性）之间的区别。当相关性是重要搜索条件的时候，应该明确指出查询字段。

_all 字段的使用需要额外的处理器周期，并且耗费更多的磁盘空间。如果不需要的话，可以完全禁用或者在每个字段的基础上自定义。

3.3.2 _field_names 字段

_field_names 字段索引文档中所有包含非空值的字段名称。_field_names 字段用于存在查询和缺失查询的情况下，查找指定字段拥有非空值的文档是否存在。

_field_name 字段的值可以用于查询、聚合以及脚本：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{"title": "This is a document"}
```

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{"title": "This is another document", "body": "This document has a body"}
```

请求：GET http://127.0.0.1:9200/secisland/_search

```

3.3.3 {
  "query": { "terms": { "_field_names": [ "title" ] } },
  "aggs": {
    "Field names": {
      "terms": { "field": "_field_names", "size": 10 }
    }
  },
  "script_fields": {
    "Field names": { "script": "doc['_field_names']" }
  }
}

```

3.3.3 _id 字段

每个被索引的文档都关联一个 `_type` 字段和一个 `_id` 字段。`_id` 字段没有索引，它的值可以从 `_uid` 字段自动生成。

`_id` 字段的值可以在查询以及脚本中访问，但是在聚合或者排序的时候，要使用 `_uid` 字段而不能用 `_id` 字段。

在查询和脚本中使用 `_id` 字段的示例如下：

请求：GET http://127.0.0.1:9200/secisland/_search

```

{
  "query": {
    "terms": { "_id": [ "1", "2" ] }
  },
  "script_fields": {
    "UID": { "script": "doc['_id']" }
  }
}

```

3.3.4 _index 字段

在多个索引中执行查询的时候，有时需要添加查询子句来关联特定的索引文档。`_index` 字段可以匹配包含某个文档的索引。在 `term` 或 `terms` 查询、聚合、脚本以及排序的时候，可以访问 `_index` 字段的值。

`_index` 是一个虚拟字段，不作为一个真实的字段添加到 Lucene 索引中。这意味着可以在 `term` 或 `terms` 查询（或任何重写 `term` 查询的查询，比如 `match`、`query_string` 或者 `simple_query_string` 查询）中使用 `_index` 字段，但是不支持 `prefix`、`wildcard`、`regexp` 或 `fuzzy` 查询。示例如下：

请求：GET http://127.0.0.1:9200/index_1,index_2/_search

```

{
  "query": { "terms": { "_index": [ "index_1", "index_2" ] } },
  "aggs": { "indices": { "terms": { "field": "_index", "size": 10 } } },
  "sort": [ { "_index": { "order": "asc" } } ],
}

```

```
"script_fields": {"index_name": {"script": "doc['_index']" }}
}
```

3.3.5 _meta 字段

每个映射类型都可以拥有自定义的元数据。这些元数据对 Elasticsearch 来说毫无用处，但是可以用来存储应用程序的特定元数据：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "user": {
      "_meta": {
        "class": "MyApp::User",
        "version": {"min": "1.0", "max": "1.3"}
      }
    }
  }
}
```

3.3.6 _parent 字段

在同一个索引中通过创建映射类型可以在文档间建立父子关系。

创建映射的代码如下：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "my_parent": {},
    "my_child": {
      "_parent": {"type": "my_parent"}
    }
  }
}
```

插入父文档的代码如下：

请求：PUT http://127.0.0.1:9200/secisland/my_parent/1

```
{"text": "This is a parent document"}
```

插入子文档，并指出父文档的代码如下：

请求：PUT http://127.0.0.1:9200/secisland/my_child/2?parent=1

```
{"text": "This is a child document"}
```

请求：PUT http://127.0.0.1:9200/secisland/my_child/3?parent=1

```
{"text": "This is another child document"}
```

1. 父子限制

父类型和子类型必须是不同的，即父子关系不能建立在相同类型的文档之间。

`_parent` 的 `type` 设置只能指向一个当前不存在的类型。这意味着一个类型被创建之后就无法成为父类型。

父子文档必须索引在相同的分片上。`parent` 编号用于作为子文档的路由值，确保子文档被索引到父文档所在的分片中。这意味着当获取、删除或更新子文档的时候，需要提供相同的 `parent` 值。

2. 整体序数

使用整体序数可以加快建立父子关系。分片发生任何改变之后，整体序数都需要进行重建。分片中存储的父编码值越多，为 `_parent` 字段重建整体序数所花的时间就越长。

整体序数在默认情况下属于懒创建：刷新之后的第一次父子查询或聚合会触发整体序数的创建，这可能会给用户的使用引入一个明显的延迟。可以使用参数将整体序数的创建时间由查询触发改到刷新触发：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "my_parent": {},
    "my_child": {
      "_parent": {
        "type": "my_parent",
        "fielddata": {"loading": "eager_global_ordinals"}
      }
    }
  }
}
```

3.3.7 _routing 字段

文档在索引中利用下面的公式路由到特定的分片：

$$\text{shard_num} = \text{hash}(\text{_routing}) \% \text{num_primary_shards}$$

`_routing` 字段的默认值使用的是文档的 `_id` 字段。如果存在父文档，则使用文档的 `_parent` 编号。

可以通过为每个文档指定一个自定义的路由值来实现自定义的路由方式：

请求：PUT `http://127.0.0.1:9200/secisland/secilog/1?routing=user1`

```
{"title": "This is a document"}
```

这个文档使用 `user1` 作为路由值，而不是它的 ID，在获取、删除和更新文档的时候需要提供相同的路由值。

`_routing` 字段可以在查询、聚合、脚本以及排序的时候访问：

请求: GET http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "terms": { "_routing": [ "user1" ] }
  },
  "aggs": {
    "Routing values": {
      "terms": { "field": "_routing", "size": 10 }
    }
  },
  "sort": [
    {
      "_routing": { "order": "desc" }
    }
  ],
  "script_fields": {
    "Routing value": { "script": "doc['_routing']" }
  }
}
```

1. 利用自定义路由进行搜索

自定义路由可以降低搜索压力。搜索请求可以仅仅发送到匹配指定路由值的分片而不是广播到所有分片,如下所示:

请求: GET http://127.0.0.1:9200/secisland/_search?routing=user1,user2

```
{
  "query": {
    "match": { "title": "document" }
  }
}
```

搜索请求仅在关联路由值为 user1 和 user2 的分片上执行。

2. 使路由值成为必选项

使用自定义路由索引、获取、删除或更新文档时,提供路由值是很重要的。

忘记路由值会导致文档被一个以上的分片索引。作为保障, `_routing` 字段可以被设置, 应使自定义路由值成为所有 CRUD 操作的必选项:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "_routing": { "required": true }
    }
  }
}
```


3. 自定义路由下的唯一编码

当索引指定了自定义路由的文档时，不能保障所有分片中文档_id的唯一性。事实上，拥有相同_id的文档会根据不同的路由存储在不同的分片中，只能依靠用户来确保编码的唯一性。

3.3.8 _source 字段

_source 字段包含索引时原始的 JSON 文档内容，字段本身不建立索引（因此无法搜索）但是会被存储，所以当执行获取请求的时候可以返回 _source 字段。虽然很方便，但是 _source 字段确实会对索引产生存储开销。因此，可以禁用 _source 字段：

请求：PUT http://127.0.0.1:9200/secilogs

```
{
  "mappings": {
    "secilog": {
      "_source": {"enabled": false}
    }
  }
}
```

如果磁盘空间是个问题，可以提高压缩等级来实现节约存储空间。可以用包含 / 排除字段的特性在保存之前减少 _source 字段的内容。



警告 如果 _source 字段被禁用，会造成大量的功能无法使用：

- ☐ 更新接口。
- ☐ 高亮显示功能。
- ☐ 重建索引的功能，不论是修改映射或分析，还是升级索引到一个新版本。
- ☐ 通过查看索引时的原始文档对查询或聚合进行调试的功能。
- ☐ 自动修复索引的功能。
- ☐ 从 _source 字段中移除内容相当于精简版的禁用功能，尤其是无法重建文档索引。

includes/excludes 参数（可以使用通配符）使用示例：

请求：PUT http://127.0.0.1:9200/logs

```
{
  "mappings": {
    "event": {
      "_source": {
        "includes": ["*.count", "meta.*"],
        "excludes": ["meta.description", "meta.other.*"]
      }
    }
  }
}
```

移除的字段不会被存储在 _source 字段中，但我们仍然可以搜索这些字段。

3.3.9 _type 字段

每个索引的文档都包含 `_type` 和 `_id` 字段, 索引 `_type` 字段的目的是通过类型名加快搜索进度。

`_type` 字段的值可以在查询、聚合、脚本以及排序时访问:

请求: PUT `http://127.0.0.1:9200/secisland/type_1/1`

```
{"text": "Document with type 1"}
```

请求: PUT `http://127.0.0.1:9200/secisland/type_2/2`

```
{"text": "Document with type 2"}
```

请求: GET `http://127.0.0.1:9200/secisland/_search/type_*`

```
{
  "query": {
    "terms": {
      "_type": [ "type_1", "type_2" ]
    }
  },
  "aggs": { "types": { "terms": { "field": "_type", "size": 10 } } },
  "sort": [
    { "_type": { "order": "desc" } }
  ],
  "script_fields": {
    "type": { "script": "doc['_type']" }
  }
}
```

3.3.10 _uid 字段

每个索引的文档都包含 `_type` 和 `_id` 字段, 这两个值结合为 `{type}#{id}` 并且作为 `_uid` 字段被索引。

`_uid` 字段的值可以在查询、聚合、脚本以及排序时访问:

请求: PUT `http://127.0.0.1:9200/secisland/secilog/1`

```
{"text": "Document with ID 1"}
```

请求: PUT `http://127.0.0.1:9200/secisland/secilog/2`

```
{"text": "Document with ID 2"}
```

请求: GET `http://127.0.0.1:9200/secisland/_search`

```
{
  "query": { "terms": { "_uid": [ "secilog#1", "secilog#2" ] } },
  "aggs": { "UIDs": { "terms": { "field": "_uid", "size": 10 } } },
  "sort": [ { "_uid": { "order": "desc" } } ],
  "script_fields": { "UID": { "script": "doc['_uid']" } }
}
```

3.4 映射参数

用于字段映射的参数，对部分或全部字段数据类型是通用的。映射参数对字段映射的特殊需求进行设置，比如设置字段的分词，字段的权重，是否被索引，日期的格式，等等，Elasticsearch 的功能非常强大，光映射参数高达 28 个，下面将一一介绍这些参数的用途。

3.4.1 analyzer 参数

可分词的字符串型字段的值通过一个分析器将字符串转换为一连串的索引词。例如，字符串“The quick Brown Foxes.”，取决于所使用的分析器，可能被分词为“quick, brown, fox”。这些是字段创建索引时的实际索引词，可以使大段文本中的特定单词的搜索更加高效。

分析的过程不仅发生在索引阶段，也发生在查询阶段：查询字符串需要通过相同（或相似）的分析器进行分析。所以，查询中的索引词和索引中索引词具有相同的格式。

Elasticsearch 附带一系列预定义的分析器，不需要更多的配置就可以使用。也附带一些字符过滤器、分词器、词元过滤器，可以结合起来为每个索引配置自定义分析器。

每个查询、每个字段或每个索引都可以指定分析器。在创建索引时，Elasticsearch 会以这个顺序查找分析器：

- ❑ 在字段映射中定义的分析器。
- ❑ 在索引设置中名为 default 的分析器。
- ❑ 标准分析器。
- ❑ 在查询时，有更多的层次。
- ❑ 在全文查询中定义的分析器。
- ❑ 在字段映射中定义的搜索分析器。
- ❑ 在字段映射中定义的分析器。
- ❑ 在索引设置中名为 default_search 的分析器。
- ❑ 在索引设置中名为 default 的分析器。
- ❑ 标准分析器。

为特定字段指定分词器最简单的方式是在字段映射中定义：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "text": {
          "type": "string",
          "fields": {
            "english": { "type": "string", "analyzer": "english" }
          }
        }
      }
    }
  }
}
```

```

    }
  }
}
}

```

search_quote_analyzer 参数设置可以对短语指定一个分析器, 当处理禁用短语连接词的查询时特别有用。

为了禁用短语连接词, 字段需要利用三个分析器设置:

- 1) analyzer 设置, 用于索引包括连接词在内的所有索引词。
- 2) search_analyzer 设置, 用于移除连接词的非短语查询。
- 3) search_quote_analyzer 设置, 用于包括连接词在内的短语查询。

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_analyzer": { "type": "custom", "tokenizer": "standard", "filter": [
          "lowercase" ] },
        "my_stop_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": [ "lowercase", "english_stop" ]
        }
      },
      "filter": {
        "english_stop": { "type": "stop", "stopwords": "_english_" }
      }
    }
  },
  "mappings": {
    "secilog": {
      "properties": {
        "title": {
          "type": "string",
          "analyzer": "my_analyzer",
          "search_analyzer": "my_stop_analyzer",
          "search_quote_analyzer": "my_analyzer"
        }
      }
    }
  }
}

```

插入数据:

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "title": "The Quick Brown Fox" }
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/2

```
{ "title": "A Quick Brown Fox" }
```

进行短语查询:

请求: GET http://127.0.0.1:9200/secisland/secilog/_search

```
{
  "query": {
    "query_string": { "query": "\"the quick brown fox\"" }
  }
}
```

当查询被包含在引号中时,表示这是一个短语查询,因此 `search_quote_analyzer` 生效并确保查询中包含连接词。如果没有 `search_quote_analyzer`,短语查询不可能被精准匹配:因为短语查询中移除了连接词,所以两个文档都会被匹配到。

3.4.2 boost 参数

在索引的时候,通过 `boost` 参数可以对一个字段进行加权:对相关性得分计数更多:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "title": { "type": "string", "boost": 2 },
        "content": {
          "type": "string"
        }
      }
    }
  }
}
```

匹配 `title` 字段会有两倍的加权,而 `content` 字段的 `boost` 参数,是默认值 1.0。

需要注意的是, `title` 字段通常会比 `content` 字段的长度短。默认的相关性计算需要考虑字段长度,因此短字段 `title` 会有一个更高的加权。



警告 索引时最好不要进行加权,主要有这几个原因:

- ❑ 除非重索引所有的文档,索引加权值不会发生改变。
- ❑ 每个查询都支持查询加权,会产生同样的效果。不同的地方在于不需要重索引就可以调整加权值。
- ❑ 索引加权值作为 `norm` 的一部分,只有一个字节。这降低了字段长度归一化因子的分辨率,会导致低质量的相关性计算。

索引加权的唯一优势在于作用于 `_all` 字段。这意味着,当查询 `_all` 字段的时候,源于 `title` 字段的词比源于 `content` 字段的词有更高的分数。这个功能是以计算成本为代价的:当索引加权被使用时,查询 `_all` 字段会变得更加缓慢。

3.4.3 coerce 参数

数据不都是干净的。一个数字取决于如何产生,可能通过 JSON 体中确定的 JSON 数值进行提供,比如 5;也可能通过一个字符串进行提供,比如“5”。或者,整型数据可能被提供浮点型数据,例如 5.0 或者“5.0”。

强制尝试清理脏值来匹配字段的数据类型。例如:

□ 字符串会被强制转换为数字。

□ 浮点型数据会被截取为整型数据。

□ 经纬地理点数据会归一化到标准 -180:180 / -90:90 坐标系统。

举例:

请求: PUT <http://127.0.0.1:9200/secisland>

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "number_one": { "type": "integer" },
        "number_two": {
          "type": "integer",
          "coerce": false
        }
      }
    }
  }
}
```

请求: PUT <http://127.0.0.1:9200/secisland/secilog/1>

```
{ "number_one": "10" }
```

请求: PUT <http://127.0.0.1:9200/secisland/secilog/2>

```
{ "number_two": "10" }
```

`number_one` 字段会包含整数 10。

因为 `number_two` 字段的强制类型转换被禁用,所以文档 2 会被丢弃。

同一个索引中相同名称的字段可以拥有不同的 `coerce` 设置。可以利用 PUT 映射接口来更新已存在字段的 `coerce` 值。

全局设置: `index.mapping.coerce` 设置可以在索引级别上对所有映射类型整体禁用强制类型转换:

请求: PUT <http://127.0.0.1:9200/secisland>

```
{
  "settings": {"index.mapping.coerce": false},
  "mappings": {
    "secilog": {
      "properties": {
        "number_one": {"type": "integer"},
        "number_two": {"type": "integer", "coerce": true}
      }
    }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "number_one": "10" }
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/2

```
{ "number_two": "10" }
```

number_one 字段继承索引级别的强制类型转换设置, number_two 字段覆盖索引级别的设置来启用强制类型转换。

3.4.4 copy_to 参数

利用 copy_to 参数可以创建自定义的 _all 字段。换句话说,多个字段的值可以被复制到一组字段,然后可以作为单个字段进行查询。例如:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "first_name": {"type": "string", "copy_to": "full_name" },
        "last_name": {"type": "string", "copy_to": "full_name" },
        "full_name": {"type": "string"}
      }
    }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{
  "first_name": "John",
  "last_name": "Smith"
}
```

请求: GET http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "match": {
```

```

    "full_name": { "query": "John Smith", "operator": "and"}
  }
}

```

first_name 和 last_name 字段的值会被复制到 full_name 字段中。first_name 和 last_name 字段仍然可以单独进行查询，而且 full_name 可以用来同时查询两个字段的內容。

一些重要的点：

- 字段的值是复制过来的，并不是分组（分析过程的结果）。
- 原始的 _source 字段不会被修改来展示复制的值。
- 利用 "copy_to": ["field_1", "field_2"], 可以将相同的值复制到多个字段。

3.4.5 doc_values 参数

大多数的字段默认被索引，使它们可以被搜索到。反向索引允许查询请求在唯一的索引词有序列表中寻找搜索的索引词，找到之后立即访问包含索引词的文档列表。

排序、聚合以及在脚本中访问字段值需要一个不同的数据访问模式。我们需要能够查找文档并在一个字段中寻找存在的索引词。

文档值（Doc Values）是磁盘上的数据结构，在文档索引阶段创建，使上面这种数据访问模式成为可能。doc_values 支持几乎所有字段类型。

所有的字段默认包含在文档值中。如果确定一个字段不需要排序、聚合或者在脚本中访问字段值，可以禁用文档值来节省存储空间：


请求：PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "status_code": { "type": "string", "index": "not_analyzed" },
        "session_id": { "type": "string", "index": "not_analyzed" },
        "doc_values": false
      }
    }
  }
}

```

status_code 字段默认拥有文档值。session_id 字段值禁用了文档值，但仍然可以被查询。

 **提示** 在同一个索引中同名的字段可以设置不同的 doc_values。可以通过 PUT 映射接口对存在的字段禁用（设置为 false）文档值。

3.4.6 dynamic 参数

默认, 字段可以被动态地添加到一个文档中, 或者添加到文档内部对象中, 仅仅通过索引一个包含新字段的文档。例如:

dynamic 设置控制新字段是否可以被动态添加, 接受三种设置, 参见如下所示:

- ☐ true——新检测到的字段会被添加到映射中(默认)。
- ☐ false——新检测到的字段会被忽略。新字段必须明确添加。
- ☐ strict——如果新字段被检测到, 一个异常会被抛出而且文档会被丢弃。

dynamic 可以在映射类型级别以及每个内部对象被设置。内部对象从它们的父对象或映射类型中继承设置值。例如:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": { "secilog": {
    "dynamic": false,
    "properties": {
      "user": {
        "properties": { "name": { "type": "string" },
          "social_networks": { "dynamic": true, "properties": {} }
        }
      }
    }
  }
}
```

动态映射在类型级别上被禁用, 所以新的顶级字段不会被自动添加。user 对象继承了类型级别的设置。user.social_networks 对象启用动态映射, 所以新字段可以被添加到这个内部对象。

3.4.7 enabled 参数

Elasticsearch 尝试索引所有的字段, 但是一些情况下仅仅需要存储字段而不进行索引。

enabled 设置, 仅可以被应用于映射类型和对象字段, 导致 Elasticsearch 跳过字段内容的分解。JSON 仍然可以从 _source 字段取回, 但是不能搜索或以其他方式存储:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "session": {
      "properties": {
        "user_id": { "type": "string", "index": "not_analyzed" },
        "last_updated": { "type": "date" },
        "session_data": { "enabled": false }
      }
    }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/session/session_1

```
{
  "user_id": "kimchy",
  "session_data": {
    "arbitrary_object": { "some_array": [ "foo", "bar", { "baz": 2 } ] }
  },
  "last_updated": "2015-12-06T18:20:22"
}
```

请求: PUT http://127.0.0.1:9200/secisland/session/session_2

```
{
  "user_id": "jpountz",
  "session_data": "none",
  "last_updated": "2015-12-06T18:22:13"
}
```

session_data 字段设置了 enabled 参数为 false, 任意的数据都可以被存储在 session_data 字段, 不过 session_data 会忽略非 JSON 对象的值。

整个映射类型都可以被禁用, 在这种情况下文档被存储在 _source 字段, 意味着它可以被取回但是没有任何内容以任何方式被索引:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "session": { "enabled": false }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/session/session_1

```
{
  "user_id": "kimchy",
  "session_data": {
    "arbitrary_object": { "some_array": [ "foo", "bar", { "baz": 2 } ] }
  },
  "last_updated": "2015-12-06T18:20:22"
}
```

整个 session 映射类型被禁用, 存储的文档可以被取回但是没有字段被添加到里面。

3.4.8 fielddata 参数

大多数的字段默认被索引, 方便搜索到。反向索引允许查询请求在唯一的索引词有序列表中寻找搜索的索引词, 找到之后立即访问包含索引词的文档列表。

排序、聚合以及在脚本中访问字段值需要一个不同的数据访问模式。我们需要能够查找

文档并在一个字段中寻找存在的索引词。

大多数的字段可以使用索引时存储在磁盘上的 `doc_values` 值来实现这种数据访问模式，但是分析过的字符串字段不支持 `doc_values`。

相对的，分词的字符串在查询时利用一个称为“字段数据”（`fielddata`）的数据结构。这个数据结构在字段被用于聚合、排序或者脚本访问的第一时间创建。从磁盘上为每个分片读取整个反向索引，倒置索引词和文档的对应关系，然后将结果存储到内存中 Java 虚拟机堆内存中，字段数据有三个参数，分别是 `format` 表示是否启用、`loading` 用的时机和 `filter` 过滤加载的数据。

加载字段数据是一个昂贵的过程，一旦被加载，就会在分片的生命周期内驻留在内存中。



警告 字段数据会消耗大量的堆内存空间，特别是加载高基数分词字符串字段。大多数时候，在分词字符串字段上进行排序或聚合没有意义（除了显著分组聚合）。经常考虑字段是否不可分词（可以使用 `doc_values`）可以更好地适应应用场景。



提示 同一个索引中的同名字段必须拥有相同的 `fielddata.*` 设置。

1. `fielddata.format`

对于分词字符串字段，字段数据 `format` 参数用来控制字段数据是否应该被启用。接受参数：`disabled` 和 `paged_bytes`（启用，默认值）。禁用字段数据加载，可以使用下面的设置：请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "text": {
          "type": "string",
          "fielddata": {"format": "disabled"}
        }
      }
    }
  }
}
```

通过上面的配置，`test` 字段不能被用于排序、聚合或者脚本引用。

2. `fielddata.loading`

这是针对每个字段的设置，控制什么时候字段数据会被加载到内存。接受三个参数，参见如下所示：

- lazy——字段数据仅在需要的时候加载到内存中（默认）。
- eager——在新的搜索分片对搜索可见之前，字段数据就被加载到内存中。如果用户搜索请求一个大容量分片，比起触发被动加载，这么做可以减少请求延迟。
- eager_global_ordinals——仅加载必须的部分到内存中。加载之后，Elasticsearch 创建整体序数数据结构来组成一个所有索引词的列表。默认情况下，整体序数是被动创建的。如果字段有非常高的基数，在这种情况下就可以使用主动加载。

3. fielddata.filter

字段数据过滤可以用来减少加载到内存中的索引词的数量，这么做可以减少内存使用量。索引词可以通过频率或正则表达式以及两者的配合进行过滤。

通过频率过滤可以加载部分索引词，这些索引词的频率落在最小值和最大值之间。频率可以表示为确切的数字（当数值比 1.0 大）或者作为一个百分比的数（例如 0.01 是 1%，1.0 是 100%）。每个分片都会计算频率。频率基于有字段值的文档数量，相对于分片中所有的文档。

可以通过 `min_segment_size` 参数指定分片应该包含的最少文件量来完全排除小容量分片：请求：`http://127.0.0.1:9200/PUT secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "tag": {
          "type": "string",
          "fielddata": {
            "filter": {
              "frequency": { "min": 0.001, "max": 0.1, "min_segment_size": 500 }
            }
          }
        }
      }
    }
  }
}
```

索引词也可以通过正则表达式进行过滤：只有匹配正则表达式的值会被加载。注意：正则表达式被应用于字段中的每个索引词，而不是整个字段值：

请求：`PUT http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "secilog": {
          "type": "string",
          "analyzer": "whitespace",
          "fielddata": {
            "filter": { "regex": { "pattern": "^#.*" } }
          }
        }
      }
    }
  }
}
```

已有的字段映射可以更新这些过滤而且会影响分片下一次加载字段数据。利用清理缓存接口来用新的过滤条件重载字段数据。

3.4.9 format 参数


在 JSON 格式文档中，日期用字符串表示。Elasticsearch 利用一系列的预先设定的格式来识别和分析这些字符串，产生一个长整型数值，代表世界标准时间的毫秒数。

除了内置格式之外，也可以使用通俗的 yyyy/MM/dd 语法来指定自定义格式：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "date": { "type": "date", "format": "yyyy-MM-dd" }
      }
    }
  }
}
```

许多支持日期值的接口也支持日期匹配表达式，例如 now-1m/d 表示当前的时间减去一个月，并且四舍五入到最近的一天。

 **提示** 在同一个索引中的同名字段必须有相同的格式设置。已有的字段可以利用 PUT 映射接口来更新格式的值。

大多数的日期都有严格的对应格式，这意味着年份、月份以及一个星期的某一天都必须用零补全格式才能有效。日期格式为 45/11/1 会被认为是无效的，需要指定完整的日期格式，例如 2045/11/01。所以需要指定格式为 strict_date_optional_time 而不是 date_optional_time。

下面列举支持的所有国际标准化组织 (ISO) 规定的日期格式，如表 3-17 所示。

表 3-17 日期格式

日期格式	说明
epoch_millis	从纪元开始的毫秒数。注意，这个时间戳的最大长度为 13 个字符，所以仅支持 1653 年到 2286 年之间的日期。在这种格式下应该使用不同的时间格式来表示日期

(续)

日期格式	说明
epoch_second	从纪元开始的秒数。注意, 这个时间戳最大长度为 10 个字符, 所以仅支持 1653 年到 2286 年之间的日期。在这种格式下应该使用不同的时间格式来表示日期
date_optional_time 或者 strict_date_optional_time	一个通用的 ISO 日期时间分析器, 日期 (date) 是必须的, 时间 (time) 是可选的
basic_date	一个完整的日期基本格式表示为四位数年份、两位数月份以及两位数当前月份的具体天数: yyyyMMdd
basic_date_time	融合基础日期和时间的基本格式, 利用 T 来分隔: yyyyMMdd 'T'HHmmss.SSSZ
basic_ordinal_date	全序日期的格式, 表示为四位数年份和三位数当前年份的具体天数: yyyyDDD
basic_ordinal_date_time	融合全序日期和时间的格式, 利用 T 来分隔: yyyyDDD 'T'HHmmss.SSSZ
basic_ordinal_date_time_no_millis	忽略毫秒数的全序日期和时间 :yyyyDDD 'T' HHmmssZ
basic_time	基础时间格式, 表示为两位当日的小时数, 两位小时里的分钟数, 两位分钟里的秒数, 三位毫秒数以及时间区间偏移: HHmmss.SSSZ
basic_time_no_millis	忽略毫秒数的基础时间格式: HHmmssZ
basic_t_time	基础时区时间格式, 表示为两位当日的小时数, 两位小时里的分钟数, 两位分钟里的秒数, 三位毫秒数以及时间区间偏移前缀 T: 'T' HHmmss.SSSZ
basic_t_time_no_millis	忽略毫秒数的基础时区时间格式: 'T'HHmmssZ
basic_week_date 或 strict_basic_week_date	全日期周年格式, 表示为四位周年数, 两位周年里的周数以及一位本周的日数: xxxx 'W' wwe
basic_week_date_time 或 strict_basic_week_date_time	融合周年和时间的格式, 利用 T 来分隔: xxxx 'W' wwe 'T' HHmmss.SSSZ
basic_week_date_time_no_millis 或 strict_basic_week_date_time_no_millis	忽略毫秒数的周年时间格式: xxxx 'W' wwe 'T' HHmmssZ
date 或 strict_date	日期格式, 表示为四位数年份, 两位数月份以及两位数当前月经过的天数: yyyy-MM-dd
date_hour 或 strict_date_hour	融合日期格式和两位当日小时数: yyyy-MM-dd 'T' HH
date_hour_minute 或 strict_date_hour_minute	融合日期格式和两位小时数, 两位分钟数: yyyy-MM-dd 'T' HH:mm
date_hour_minute_second 或 strict_date_hour_minute_second	融合日期格式和两位小时数, 两位分钟数以及两位秒数: yyyy-MM-dd 'T' HH:mm:ss
date_hour_minute_second_fraction 或 strict_date_hour_minute_second_fraction	融合日期格式和两位小时数, 两位分钟数, 两位秒数以及三位毫秒部分: yyyy-MM-dd 'T' HH:mm:ss.SSS
date_hour_minute_second_millis 或 strict_date_hour_minute_second_millis	融合日期格式和两位小时数, 两位分钟数, 两位秒数以及三位毫秒部分: yyyy-MM-dd 'T' HH:mm:ss.SSS
date_time 或 strict_date_time	融合全日期和时间, 利用 T 分隔: yyyy-MM-dd 'T' HH:mm:ss.SSSZZ

(续)

日期格式	说明
date_time_no_millis 或 strict_date_time_no_millis	融合日期和时间, 忽略毫秒数, 利用 T 分隔: yyyy-MM-dd 'T' HH:mm:ssZZ
hour 或 strict_hour	当日两位小时数的格式: HH
hour_minute 或 strict_hour_minute	两位小时数, 两位分钟数: HH:mm
hour_minute_second 或 strict_hour_minute_second	两位小时数, 两位分钟数, 两位秒数: HH:mm:ss
hour_minute_second_fraction 或 strict_hour_minute_second_fraction	两位小时数, 两位分钟数, 两位秒数, 三位毫秒数: HH:mm:ss.SSS
hour_minute_second_millis 或 strict_hour_minute_second_millis	两位小时数, 两位分钟数, 两位秒数, 三位毫秒数: HH:mm:ss.SSS
ordinal_date 或 strict_ordinal_date	全序日期格式, 表示为四位数年份和三位当年已过的天数: yyyy-DDD
ordinal_date_time 或 strict_ordinal_date_time	融合全序日期和时间: yyyy-DDD 'T' HH:mm:ss.SSSZZ
ordinal_date_time_no_millis 或 strict_ordinal_date_time_no_millis	忽略毫秒数的全序日期和时间: yyyy-DDD 'T' HH:mm:ssZZ
time 或 strict_time	时间格式, 表示为两位当日的小时数, 两位小时里的分钟数, 两位分钟里的秒数, 三位毫秒数以及时间区间偏移: HH:mm:ss.SSSZZ
time_no_millis 或 strict_time_no_millis	忽略毫秒数的时间格式: HH:mm:ssZZ
t_time 或 strict_t_time	时区时间格式, 表示为两位当日的小时数, 两位小时里的分钟数, 两位分钟里的秒数, 三位毫秒数以及时间区间偏移前缀 T: 'T' HH:mm:ss.SSSZZ
t_time_no_millis 或 strict_t_time_no_millis	忽略毫秒数的时区时间格式: 'T' HH:mm:ssZZ
week_date 或 strict_week_date	全周年日期表示为四位周年数, 两位当年的周数, 以及一位本周的日数: xxxx-'W' ww-e
week_date_time 或 strict_week_date_time	融合全周年日期和时间, 利用 T 分隔: xxxx-'W'ww-e 'T' HH:mm:ss.SSSZZ
weekyear 或 strict_weekyear	周年格式, 表示为四位周年数: xxxx
weekyear_week 或 strict_weekyear_week	融合周年格式和两位周年的周数: xxxx-'W'ww
weekyear_week_day 或 strict_weekyear_week_day	融合周年格式和两位周年的周数, 以及一位本周的日数: xxxx-'W'ww-e
year 或 strict_year	四位年数的格式: yyyy
year_month 或 strict_year_month	四位年数和两位月数: yyyy-MM
year_month_day 或 strict_year_month_day	四位年数, 两位月数, 两位日数: yyyy-MM-dd

3.4.10 geohash 参数

地理散列是把地球划分为网格的经纬度编码。网格的每个单元格被表现为地理散列字符串。每个单元格可以被进一步划分为用更长字符串表示的更小单元格。所以地理散列越长,

(续)

单元格越小(精确度越高)。

因为地理散列就是个字符串,可以像其他字符串一样被存储在一个反向索引中,这使得索引地理散列效率很高。

如果启用地散列选项,一个地理散列类型的“子字段”会被索引,例如 .geohash。地理散列的长度通过 geohash_precision 参数控制。

如果启用 geohash_prefix 选项,geohash 选项会被自动启用。

例如:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "location": { "type": "geo_point", "geohash": true }
      }
    }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{
  "location": { "lat": 41.12, "lon": -71.34 }
}
```

每个地理点都会索引一个 location.geohash 字段。

地理散列可以从 doc_values 中取回。

3.4.11 geohash_precision 参数

当启用地散列选项的时候,地理精度设置可以控制地理散列的长度;当地理散列前缀选项启用的时候,地理精度设置可以控制地理散列的最大长度。

接受参数:

- ☐ 介于 1 到 12 (默认) 之间的数字,代表地理散列的长度。
- ☐ 一段距离,例如 1km。如果指定了一段距离,会被作为最小的地理散列长度提供请求分辨率。

例如:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "location": { "type": "geo_point", "geohash_precision": true,

```

```

    "geohash_precision": 6 }
  }
}
}

```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```

{
  "location": {"lat": 41.12, "lon": -71.34}
}

```

一个 geohash_precision 值为 6 的地理散列单元格的面积近似为 1.26km x 0.6km。

3.4.12 geohash_prefix 参数

当启用地理散列选项, 以一定的精度将地理散列作为经纬度点索引的时候, geohash_prefix 选项也会索引到所有的地理散列单元格中。

例如, 一个 drm3btev3e86 地理散列会索引下列所有的索引词: [d, dr, drm, drm3, drm3b, drm3bt, drm3bte, drm3btev, drm3btev3, drm3btev3e, drm3btev3e8, drm3btev3e86]。

在 geohash_cell 查询中可以使用地理散列前缀来查找特定地理散列中的点, 或者它的相邻点:

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "location": {"type": "geo_point", "geohash_prefix": true,
          "geohash_precision": 6}
      }
    }
  }
}

```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```

{
  "location": {"lat": 41.12, "lon": -71.34}
}

```

请求: GET http://127.0.0.1:9200/secisland/_search?fielddata_fields=location.geohash

```

{
  "query": {
    "geohash_cell": {
      "location": {"lat": 41.02, "lon": -71.48},
      "precision": 4,
      "neighbors": true
    }
  }
}

```

```
}
}
```

3.4.13 ignore_above 参数

比 ignore_above 设置长的字符串不会被分词或者索引。这主要用于不分词的字符串字段，这些字符串通常用来过滤、聚合以及排序。这些都是结构化的字段，让这些字段索引非常长的索引词通常是不明智的。

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "message": { "type": "string", "index": "not_analyzed", "ignore_above": 20 }
      }
    }
  }
}
```

message 字段会忽略任何长度超过 20 个字符的字符串。

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{"message": "Syntax error"}
```


这篇文档会被成功索引。

请求: PUT http://127.0.0.1:9200/secisland/secilog/2

```
{"message": "Syntax error with some long stacktrace"}
```

这篇文档会被索引，但是不会索引 message 字段的值。

这个选项也可以用于防止 Lucene 限制索引词的字节长度为 32766。

 **注意** ignore_above 的值是字符数，而 Lucene 是字节计数。如果使用 UTF-8 文本与许多非 ASCII 字符，可能需要设置限制为 $32766 / 3 = 10922$ 。因为 UTF-8 字符会占用最多 3 个字节。

3.4.14 ignore_malformed 参数

通常对于收到的数据没有做更多的控制。一个用户发送的 login 字段可能是日期，另一个发送的 login 字段可能是电子邮件地址。

默认情况下，在试着索引错误的数据类型的时候会抛出异常并拒绝整个文档。如果 ignore_malformed 参数被设置为 true，异常会被忽略。错误字段不会被索引，但文档中的其他字段会正常处理。

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "number_one": {"type": "integer"},
        "number_two": {"type": "integer", "ignore_malformed": true}
      }
    }
  }
}
```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "text": "Some text value", "number_one": "foo" }
```

这篇文档会被拒绝索引。

请求: PUT http://127.0.0.1:9200/secisland/secilog/2

```
{ "text": "Some text value", "number_two": "foo" }
```

这篇文档会索引 text 字段:

❑ index-level 默认

❑ index.mapping.ignore_malformed 设置可以用在索引级别上, 在所有映射类型中忽略错误内容:

请求: PUT http://127.0.0.1:9200/secisland

```
{
  "settings": { "index.mapping.ignore_malformed": true },
  "mappings": {
    "secilog": {
      "properties": {
        "number_one": { "type": "byte" },
        "number_two": { "type": "integer", "ignore_malformed": false }
      }
    }
  }
}
```

3.4.15 include_in_all 参数

include_in_all 参数对每个字段进行控制是否被包含在 _all 字段中。默认值为 true, 除非索引被设为 no。

include_in_all 参数还可以被设置在类型级别以及对象或嵌入字段, 所有的子字段会继承这个设置, 例如:

请求: PUT http://127.0.0.1:9200/secisland

```
{
```

```

"mappings": {
  "secilog": {
    "include_in_all": false,
    "properties": {
      "title": { "type": "string" },
      "author": {
        "include_in_all": true,
        "properties": { "first_name": { "type": "string" },
          "last_name": { "type": "string" } }
      },
      "editor": {
        "properties": { "first_name": { "type": "string" },
          "last_name": { "type": "string", "include_in_all": true } }
      }
    }
  }
}

```

3.4.16 index 参数

index 选项控制字段值如何进行索引，以及如何搜索。接受如下三种值：

- no——不要在索引中加入这个字段的值。利用这个设置，字段不会被查询到。
- not_analyzed——字段值原封不动地添加到索引中，作为单一索引词。除了字符串字段之外的所有字段默认支持这个选项。
- analyzed——这个选项仅仅用在字符串字段上，作为默认选项。字符串值首先被分词为一组索引词，然后被索引。在搜索的时候，查询字符串会通过相同的分词器生成相同格式的索引词。正是由于这个过程，使得全文搜索成为可能。

举例，创建一个不参与分词的字符串字段：

请求：PUT <http://127.0.0.1:9200/secisland>

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "status_code": { "type": "string", "index": "not_analyzed" }
      }
    }
  }
}

```

3.4.17 index_options 参数

index_options 参数控制将什么信息添加到反向索引，用于搜索和强调的目的。接受下面的参数：

- docs——只有被索引的文档数量。可以解决“字段中是否包含这个索引词”的问题。

- freqs——被索引的文档数量和索引词频率。索引词频率用来使重复索引词的得分高于单个索引词。
- positions——文档数量，索引词频率以及索引词位置。位置可以被用于邻近或短语查询。
- offsets——文档数量，索引词频率、位置以及开始和结束字符偏移量（映射索引词到原始字符串）。

分词字符串字段利用 positions 作为默认值，其他字段利用 docs 作为默认值。

请求：PUT <http://127.0.0.1:9200/secisland>

```
{
  "mappings": {
    "secilog": {"properties": {"text": {"type": "string",
      "index_options": "offsets"}}}
  }
}
```

3.4.18 lat_lon 参数

地理查询通过添加每个地理点字段的值到一个公式中，执行这个公式来决定地理点是否落入请求的区域中。不像大多数查询，反向索引并不复杂。

设置 lat_lon 为 true 会使纬度和经度的值作为数字型字段（称为 .lat 和 .lon）被索引。这些字段可以用于地理范围查询和地理距离查询代替执行内存运算。

请求：PUT <http://127.0.0.1:9200/secisland>


```
{
  "mappings": {
    "secilog": {
      "properties": {
        "location": {"type": "geo_point", "lat_lon": true}
      }
    }
  }
}
```

设置 lat_lon 为 true，会在 location.lat 和 location.lon 字段索引地理点。

请求：GET http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "geo_distance": {
      "location": {"lat": 41, "lon": -71},
      "distance": "50km",
      "optimize_bbox": "indexed"
    }
  }
}
```

`indexed` 参数会让地理距离查询利用反向索引值而不是内存运算。
是执行内存还是索引操作更好，取决于数据集和执行的查询类型。

 **注意** `lat_lon` 选项仅对单值的地理点字段有意义。不会对地理点数组产生效果。

3.4.19 fields 参数

多字段的目的是基于不同的目的用不同的方法索引相同的字段。例如，一个字符串字段可以作为分词字段被索引用于全文搜索，也可以作为不可分词字段用于排序或聚合：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "city": {
          "type": "string",
          "fields": {
            "raw": { "type": "string", "index": "not_analyzed" }
          }
        }
      }
    }
  }
}
```

`city.raw` 字段是 `city` 字段不可分词的版本。`city` 字段是分词字段，可以用来进行全文搜索。`city.raw` 字段可以用来排序或聚合。

使用多重分词器：另一种多字段的情况是用不同的方式对相同的字段进行分词来达到更好的相关性。例如，我们可以索引一个字段，利用标准分词器将文本划分为单词，然后利用英文分词器把单词划分为它们的词根形式：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "text": {
          "type": "string",
          "fields": {
            "english": { "type": "string", "analyzer": "english" }
          }
        }
      }
    }
  }
}
```

```
}
}
```

text 字段使用标准分词器，text.english 字段使用英文分词器。

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "text": "quick brown fox" }
```

请求：PUT http://127.0.0.1:9200/secisland/secilog/2

```
{ "text": "quick brown foxes" }
```

索引两个文档，第一个文档中的 text 字段包含 fox 索引词，第二个文档中的 text 字段包含 foxes 索引词。两个文档的 text.english 字段均包含 fox 索引词，因为 foxes 的词根是 fox。

3.4.20 norms 参数

norms 存储各种标准化系数（一个数字），表示相关字段的长度和索引时相关性加权设置。将会用在查询的时候计算文档对于查询条件的相关性得分。

虽然对于计算相关性得分非常有用，但是 norms 也需要大量的内存。所以，如果不需要在一个特殊的字段上计算相关性得分，应该在字段上禁用 norms。在这种特别的情况下，字段仅仅用于过滤或聚合。

norms 可以利用 PUT 映射接口取消（不能被重新启用）：

请求：PUT http://127.0.0.1:9200/secisland/_mapping/secilog

```
{
  "properties": {
    "title": {
      "type": "string",
      "norms": {"enabled": false}
    }
  }
}
```



注意 标准值不会立即移除，但是索引新文件、旧分片融入新分片时，标准值会被移除。因为一些文档没有标准值，另外的仍然有标准值，任何移除已有标准值的字段计算出的得分可能会返回不同的结果。

标准值延迟加载：当新分片上线，标准值可以被优先（eager）加载到内存，或者当字段被查询时，标准值被延迟加载（lazy，默认）。

请求：PUT http://127.0.0.1:9200/secisland/_mapping/secilog

```
{
  "properties": {
    "title": {
```

```

    "type": "string",
    "norms": { "loading": "eager" }
  }
}

```

3.4.21 null_value 参数

空值是不能被索引或搜索的。当一个字段设置为 null (或者是一个空数组或者 null 值的数组), 这个字段会当作没有值的字段。

null_value 参数可以用指定的值替换掉确切的空值, 以便可以被索引和搜索。例如:

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "status_code": { "type": "string", "index": "not_analyzed",
          "null_value": "NULL" }
      }
    }
  }
}

```

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```

{"status_code": null}

```

利用索引词 NULL 替换掉确切的 null 值。

请求: PUT http://127.0.0.1:9200/secisland/secilog/2

```

{"status_code": [] }

```

因为空数组不包含确切的 null 值, 所以不会被 null_value 的值替换。



注意 null_value 需要设置和字段相同的数据类型。

3.4.22 position_increment_gap 参数

为了可以支持短语查询, 需要保存可分词字符串字段中分词的位置。当字符串字段索引多个值, 一个“虚拟”缺口会被加到各个值之间来防止短语查询跨值匹配。缺口的大小可以利用 position_increment_gap 配置, 默认值是 100。

例如:

请求: PUT http://127.0.0.1:9200/secisland

```

{

```

```

"mappings": {
  "secilog": {
    "properties": {
      "names": { "type": "string", "position_increment_gap": 0 }
    }
  }
}

```

请求: PUT <http://127.0.0.1:9200/secisland/secilog/1>

```
{ "names": [ "John Abraham", "Lincoln Smith" ] }
```

下一个元素数组的第一个索引词与上一个元素数组的最后一个索引词之间会有 0 个索引词间隔。

请求: POST http://127.0.0.1:9200/secisland/secilog/_search

```

{
  "query": {
    "match_phrase": { "names": "Abraham Lincoln" }
  }
}

```

短语查询匹配文档是怪异的,但是这正是我们在映射中要求的。

3.4.23 precision_step 参数

大多数的数字型数据类型索引额外的索引词表示每个数字的范围,使范围查询更加便捷:

```

"range": {
  "number": { "gte": 0 "lte": 321 }
}

```

这从本质上作为索引词查询,结构如下:

```

"terms": {
  "number": [ "0-255", "256-319" "320", "321" ]
}

```

precision_step 的默认值取决于数字型字段的类型。

3.4.24 properties 参数

类型映射,对象字段和嵌套类型字段包含子字段,称为属性。这些属性可以是任何数据类型,包含对象和嵌套类型。属性在以下情况中添加:

- ☐ 创建索引的时候明确定义。
- ☐ 利用创建映射接口添加或修改映射类型的时候明确定义。
- ☐ 索引包含新字段的文档可以动态添加。

向映射类型中添加属性,包含一个对象字段和嵌套类型字段的例子:


请求: PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "manager": {
          "properties": { "age": { "type": "integer" },
            "name": { "type": "string" } }
        },
        "employees": {
          "type": "nested",
          "properties": { "age": { "type": "integer" },
            "name": { "type": "string" } }
        }
      }
    }
  }
}
```

点符号: 查询、聚合等方法可以使用点符号获取内部字段:

请求: GET http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "match": { "manager.name": "Alice White" }
  },
  "aggs": {
    "Employees": {
      "nested": { "path": "employees" },
      "aggs": {
        "Employee Ages": {
          "histogram": { "field": "employees.age", "interval": 5 }
        }
      }
    }
  }
}
```

 **注意** 必须指定内部字段的完整路径。

3.4.25 search_analyzer 参数

通常,索引时和搜索时应该使用相同的分词器,确保查询时的索引词和反向索引中的索引词有相同的格式。

但有些时候,在搜索时使用不同的分词器是有意义的,比如当使用 edge_ngram 标记器来自动完成。

默认情况下, 查询会使用字段映射中定义的分词器, 但是可以利用 `search_analyzer` 设置来重写配置:

请求: PUT <http://127.0.0.1:9200/secisland>

```
{
  "settings": {
    "analysis": {
      "filter": {
        "autocomplete_filter": {"type": "edge_ngram", "min_gram": 1, "max_gram": 20}
      },
      "analyzer": {
        "autocomplete": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "autocomplete_filter"]
        }
      }
    }
  },
  "mappings": {
    "secilog": {
      "properties": {
        "text": {"type": "string", "analyzer": "autocomplete",
          "search_analyzer": "standard"}
      }
    }
  }
}
```

分词器设置定义了自定义的自动完成分词器。

`text` 字段在索引的时候使用自动完成分词器, 在搜索的时候使用标准分词器。

3.4.26 similarity 参数

Elasticsearch 允许对每个字段配置得分算法或者相似算法。similarity 参数提供了一个简单的方式来选择不同于默认的 TF/IDF 相似算法, 例如 BM25。

相似算法多数用于字符串字段, 特别是可分词的字符串字段, 但是也可以应用于其他数据类型。

自定义相似算法可以通过调整内置相似算法的参数进行配置。

不用更多配置就可以直接使用的相似算法只有:

❑ default: Elasticsearch 和 Lucene 默认使用的 TF/IDF 算法。

❑ BM25: Okapi BM25 算法。

相似算法可以在字段第一次创建的时候在字段级别上进行设置:

请求: PUT <http://127.0.0.1:9200/secisland>

```
{
```

```

"mappings": {
  "secilog": {
    "properties": {
      "default_field": {
        "type": "string"
      },
      "bm25_field": { "type": "string", "similarity": "BM25" }
    }
  }
}

```

3.4.27 store 参数

默认情况下, 字段值被索引来确保可以被搜索, 但是不会被存储。这意味着可以查询字段, 但是无法取回原始字段值。

通常这没有什么问题。字段值早已是默认存储的 `_source` 字段的一部分。如果仅仅想取回单个字段或一些字段的值, 而不是整个 `_source` 字段, 可以通过数据源过滤来实现:

请求: `PUT http://127.0.0.1:9200/secisland`

```

{
  "mappings": {
    "secilog": {
      "properties": {
        "title": { "type": "string", "store": true },
        "date": { "type": "date", "store": true },
        "content": { "type": "string" }
      }
    }
  }
}

```


`title` 和 `date` 字段会被存储。

```

POST secisland/_search
{"fields": [ "title", "date" ] }

```

这个请求可以取回 `title` 和 `date` 的字段值。

 **注意** 为了保持一致性, 存储的字段作为一个数组返回。因为没有办法知道原始的字段值是一个单值、多值还是个空数组。如果需要原始值, 应该从 `_source` 字段中取回。

3.4.28 term_vector 参数

索引词向量包含分析过程产生的索引词信息, 包括:

□ 索引词列表。

- ☐ 每个索引词的位置（或顺序）。
- ☐ 映射索引词到原始字符串中的原始位置中开始和结束字符的偏移量。

这些索引词向量会被存储，所以可以作为一个特殊文档取回。

term_vector 设置接受以下参数：

- ☐ no——不存储索引词向量（默认）。
- ☐ yes——只存储字段的索引词。
- ☐ with_positions——索引词和位置将会被存储。
- ☐ with_offsets——索引词和字符偏移量会被存储。
- ☐ with_positions_offsets——索引词，位置以及字符偏移量都会被存储。



警告 设置 with_positions_offsets 会使字段索引的大小翻倍。

映射示例：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "properties": {
        "text": { "type": "string", "term_vector": "with_positions_offsets" }
      }
    }
  }
}
```

3.5 动态映射

在 Elasticsearch 中可以不事先建好索引结构，在使用的时候可以直接插入文档到索引中，系统会根据文档的内容自动进行索引结构的动态映射，这样就极大地简化了索引的操作。

3.5.1 概念

Elasticsearch 最重要的一个特征就是尝试跳出固有的方式，使用户尽可能快地搜索数据。索引一篇文档，不需要事先创建索引、定义映射类型以及字段，仅需要直接索引一篇文档，然后索引、类型和字段会自动生成：

```
PUT data/counters/1
{ "count": 5 }
```

自动创建索引为 data，映射类型为 counters，字段名为 count，类型为长整型（long）

自动检测添加新类型和字段，被称为动态映射。可以根据目的自定义动态映射的规则：

□ `_default` 映射：用于创建新映射类型的基础映射。

□ 动态字段映射：控制动态字段检测规则。

□ 动态模板：自定义规则来配置动态添加字段的映射。

禁用动态类型创建：动态类型创建可以通过设置 `index.mapper.dynamic` 为 `false` 来禁用，无论是在配置文件 `config/elasticsearch.yml` 中进行设置还是在每个索引中进行设置：

请求：PUT `http://127.0.0.1:9200/_settings`

```
{ "index.mapper.dynamic": false }
```

对所有的索引禁用动态类型创建。

无论设置的值是什么，映射类型仍然可以在创建索引或者使用添加映射接口显式添加。

3.5.2 `_default` 映射

默认映射，用作每个新映射类型的基础映射，可以通过在索引中增加名为 `_default` 的映射类型来自定义：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "_default_": {
      "_all": { "enabled": false }
    },
    "user": { },
    "blogpost": {
      "_all": { "enabled": true }
    }
  }
}
```

□ `_default` 映射禁用了 `_all` 字段。

□ `user` 映射从 `_default` 映射中继承了设置。

□ `blogpost` 映射重写了默认设置，启用了 `_all` 字段。

`_default` 映射可以在索引创建之后进行修改，新的默认映射只会影响到之后创建的映射类型。

3.5.3 动态字段映射

默认情况下，在文档中发现新字段的时候，Elasticsearch 会添加新字段到类型映射中。这种行为可以被禁用，通过设置动态参数为 `false` 或 `strict`。

如果动态字段映射被启用，有一些简单的规则用于决定字段应该是哪一种数据类型，如表 3-18 所示。

表 3-18 动态字段映射对应的数据类型

JSON 数据类型	Elasticsearch 数据类型
null	不添加字段
true 或 false	boolean 类型字段
浮点型数字	双精度浮点型字段
整数	长整型字段
对象	对象类型字段
数组	取决于数组中第一个非空值的类型
字符串	可能是日期型字段（值通过日期检查），双精度或长整型字段（值通过数字检查），可分词字符串型字段

这些是仅有的可以动态添加的字段数据类型。所有其他的数据类型都必须被明确地添加到映射中。

除了下面列出的选项，可以通过动态模板自定义更多的动态字段映射规则。

1. 日期检查

如果启用日期检查（默认），新的字符串字段的内容会被检查是否匹配任何 `dynamic_date_formats` 中指定的日期格式。如果发现匹配，新的日期字段会以相同的格式添加。

`dynamic_date_formats` 默认值是：

```
[ "strict_date_optional_time", "yyyy/MM/dd HH:mm:ss Z||yyyy/MM/dd Z"]
```

2. 禁用日期检查

通过设置 `date_detection` 的值为 `false` 来禁用动态日期检查：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {"date_detection": false}
  }
}
```

3. 自定义日期检查格式

动态日期格式可以自定义：

请求：PUT `http://127.0.0.1:9200/secisland`

```
{
  "mappings": {
    "secilog": {"dynamic_date_formats": ["MM/dd/yyyy"]}
  }
}
```

4. 数字检查

因为 JSON 支持浮点型和整型数据类型，一些应用或语言有时会作为字符串传递数字。

通常的解决方式是明确定义这些字段映射，但是可以启用数字检查（默认禁用）来自动做类型转换：

请求：PUT <http://127.0.0.1:9200/secisland>

```
{
  "mappings": {
    "secilog": {"numeric_detection": true}
  }
}
```

3.5.4 动态模板

动态模板可以定义自定义映射用来动态添加字段，基于以下参数：

- ❑ 通过 Elasticsearch 进行数据类型检查，利用 `match_mapping_type` 参数。
- ❑ 字段名，利用 `match` and `unmatch` 或 `match_pattern` 参数。
- ❑ 字段全路径，利用 `path_match` and `path_unmatch` 参数。

原始字段名 `{name}` 和数据类型检查 `{dynamic_type}` 模板变量可以作为占位符用在映射标准中。



注意 动态字段映射仅在字段包含具体值（不为 `null` 或空数组）的时候添加。这意味着如果在 `dynamic_template` 中有 `null_value` 选项，第一个文档中被索引的字段有具体的值之后，`null_value` 才会起作用。

动态模板被指定为命名的对象数组：

```
"dynamic_templates": [
  {
    "my_template_name": {
      ... match conditions ...
      "mapping": { ... }
    }
  },
  ...
]
```

- ❑ 模板名可以是任何字符串值。
- ❑ 匹配条件可以包含所有的：`match_mapping_type`、`match`、`match_pattern`、`unmatch`、`path_match`、`path_unmatch`。
- ❑ `mapping` 包含匹配到的字段需要使用的映射值。

模板顺序执行——第一个匹配到的模板起作用。新的模板可以用增加映射接口添加到列表的末尾。如果新的模板和已经存在的模板有相同的名字，旧版本会被替换掉。

1. match_mapping_type

match_mapping_type 匹配数据类型，通过动态字段映射检查，换句话说就是 Elasticsearch 认为字段需要拥有的数据类型。只有下面的数据类型可以被自动检查：boolean, date, double, long, object, string。也可以接受 * 来匹配所有的数据类型。

举个例子，我们需要映射所有整数型字段为 integer 而不是 long，以及所有的可分词和不可分词的字符串型字段，我们可以使用下面的模板：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "dynamic_templates": [
        {
          "integers": {
            "match_mapping_type": "long",
            "mapping": { "type": "integer" }
          },
          "strings": {
            "match_mapping_type": "string",
            "mapping": {
              "type": "string",
              "fields": {
                "raw": { "type": "string", "index": "not_analyzed",
                  "ignore_above": 256 }
              }
            }
          }
        }
      ]
    }
  }
}
```

2. match 和 unmatched

match 参数使用匹配字段名称的方式，unmatch 使用排除 match 匹配的字段的方式。

举个例子，匹配所有字符串型字段名字以 long_ 开头并排除以 _text 结束的字段，索引这些字段为长整型字段：

请求：PUT http://127.0.0.1:9200/secisland

```
{
  "mappings": {
    "secilog": {
      "dynamic_templates": [
        {
          "longs_as_strings": {

```

```

    "match_mapping_type": "string",
    "match": "long_*",
    "unmatch": "*_text",
    "mapping": {"type": "long"}
  }
}
}
}
}

```

3. match_pattern

match_pattern 参数支持完整的 Java 正则表达式匹配字段名而不是简单的通配符:

```
"match_pattern": "regex"
```

4. path_match 和 path_unmatch

工作方式与 match 和 unmatch 相同,但是在字段全路径上进行操作,不仅仅是在最终的名字上:

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "dynamic_templates": [
        {
          "full_name": {
            "path_match": "name.*",
            "path_unmatch": "*_middle",
            "mapping": {"type": "string", "copy_to": "full_name"}
          }
        ]
      }
    }
  }
}

```

5. {name} 和 {dynamic_type}

占位符会被映射中的字段名和字段类型替换。举个例子,设置所有的字符型字段使用与字段同名的分析器,禁用所有非字符型字段的 doc_values 参数:

请求: PUT http://127.0.0.1:9200/secisland

```

{
  "mappings": {
    "secilog": {
      "dynamic_templates": [
        {
          "named_analyzers": {

```

```

    "match_mapping_type": "string",
    "match": "*",
    "mapping": { "type": "string", "analyzer": "{name}" }
  },
  {
    "no_doc_values": {
      "match_mapping_type": "*",
      "mapping": { "type": "{dynamic_type}", "doc_values": false }
    }
  }
]
}
}
}

```

3.5.5 重写默认模板

可以重写所有索引的默认映射，也可以通过在索引模板中指定 `_default` 类型映射重写映射类型。

例如，为了对所有新索引中的类型禁用 `_all` 字段，可以创建下面这样的索引模板：

请求：PUT http://127.0.0.1:9200/_template/disable_all_field

```

{
  "disable_all_field": {
    "order": 0,
    "template": "*",
    "mappings": {
      "_default_": {
        "_all": { "enabled": false }
      }
    }
  }
}
}

```

3.6 小结

本章介绍了 Elasticsearch 的映射，映射是 Elasticsearch 内部结构对外的一个展现方式，通过本章的学习可以了解 Elasticsearch 所支持字段的数据类型，以及 Elasticsearch 内置的元数据知识；每一个类型都支持很多的参数，通过参考可以控制 Elasticsearch 的很多行为，比如是否索引等；同时 Elasticsearch 也是非常智能的，默认情况下，映射可以自动创建，并且可以很好地工作。

第4章 Chapter 4

搜索

前面的章节主要介绍了Elasticsearch的索引和映射，但这两个内容只解决存储的问题，在使用Elasticsearch的时候，大部分是和搜索相关的内容。本章开始介绍Elasticsearch的搜索功能，通过本章的学习可以详细了解搜索的语法和查询专用语言(DSL)的相关知识。

4.1 深入搜索

之前简单介绍过搜索，在这里对搜索进行详细的介绍。包括搜索详细参数，利用重新评分对短语进行搜索，通过滚动查询对索引进行全量搜索，对索引文档中的隐藏参数进行查询，介绍在搜索中用到的常用函数和搜索模板。

4.1.1 搜索方式

从2.7节可以知道，搜索有两种方式，一种是通过URL参数进行搜索，另一种是通过POST请求参数进行搜索。下面详细介绍这两种方式，以及相关的几个内容，如排序、数据过滤、脚本支持。

1. URL 参数搜索

请求：GET http://127.0.0.1:9200/secisland/log/_search? 参数，多个参数用&分开。参数的解释参见表4-1。

表 4-1 URL 搜索参数

参数	解释
q	查询字符串, 例如: q=syslog
df	当查询中没有定义前缀的时候默认使用的字段
analyzer	当分析查询字符串的时候使用的分词器
lowercase_expanded_terms	搜索的时候忽略大小写标志, 默认为 true
analyze_wildcard	通配符或者前缀查询是否被分析, 默认为 false
default_operator	默认多个条件的关系, AND 或者 OR, 默认为 OR
lenient	如果设置为 true, 字段类型转换失败的时候将被忽略, 默认为 false
explain	在每个返回结果中, 将包含评分机制的解释
_source	是否包含元数据, 同时支持 _source_include 和 _source_exclude
fields	只返回索引中指定的列, 多个列中间用逗号分开
sort	根据字段名排序, 例如 fieldName:asc 或者 fieldName:desc
track_scores	评分轨迹, 当排序的时候, true 表示返回评分的信息
timeout	超时的时间设置
terminate_after	在每个分片中查询的最大条数, 如果设置, 返回结果中会有一个 terminated_early 字段
from	返回的索引匹配结果的开始值, 默认为 0
size	搜索结果返回的条数, 默认为 10
search_type	搜索的类型, 可以是 dfs_query_then_fetch、query_then_fetch, 默认为 query_then_fetch

2. POST 请求参数搜索

请求: POST http://127.0.0.1:9200/secisland/log/_search。参数在请求头中。

参数是 JSON 格式的查询领域语法 (query dsl)。例如:

```
{
  "query" : { "term" : { "type" : "syslog" } }
}
```


返回值:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
```

```
{
  "_index" : "secilog",
  "_type" : "log",
  "_id" : "1",
  "_score" : 1.0,
  "_source" : { "type" : "syslog", "message" : "secilog test test test " }
}
```

如果需要搜索分页，可以通过 from size 组合来进行。from 表示从第几行开始，size 表示查询多少条文档。from 默认为 0，size 默认为 10，例如：

```
{
  "from" : 0, "size" : 10,
  "query" : {
    "term" : { "type" : "syslog" }
  }
}
```

 size 的大小不能超过 index.max_result_window 这个参数的设置，默认为 10 000。

3. 排序 sort

当搜索的字段有多个时，可以对指定字段进行排序。例如下面的搜索优先对 `type` 字段进行排序，然后对 `message` 字段进行排序：

```
{
  "sort" : [
    { "type" : { "order": "asc" } },
    { "message" : { "order": "asc" } },
  ]
}
```

返回值:

```
{
  "took" : 2,
  "timed_out" : false,
  "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 },
  "hits" : {
    "total" : 2,
    "max_score" : null,
    "hits" : [ {
      "_index" : "secilog",
      "_type" : "log",
      "_id" : "2",
      "_score" : null,
      "_source" : { "type" : "file", "message" : "Another secilog test " },
      "sort" : [ "file", "test" ]
    } ]
  }
}
```

```

    }, {
      "_index" : "secilog",
      "_type" : "log",
      "_id" : "1",
      "_score" : null,
      "_source" : { "type" : "syslog", "message" : "secilog test test test " },
      "sort" : [ "syslog", "test" ]
    } ]
  }
}

```

当一个字段的内容有多个值的时候，系统支持一些计算进行排序，包括 min、max、sum、avg、median（中间值）。例如下面的请求表示 order 有多个值，取平均值排序的方式如下：

```

{
  "query" : {
    ...
  },
  "sort" : [{"price" : { "order" : "asc", "mode" : "avg" }}]
}

```

4. 数据列过滤

数据列过滤允许在查询的时候不显示原始数据，或者显示部分原始字段。例如不显示原始文档的方式如下：

```

{
  "_source": false,
  "query" : { "term" : { "type" : "syslog" } }
}

```

显示部分文档列的方式如下：

```

{
  "_source": "obj.*",
  "query" : { "term" : { "type" : "syslog" } }
}
{
  "_source": [ "obj1.*", "obj2.*" ],
  "query" : { "term" : { "type" : "syslog" } }
}

```

还可以包含或者排除某些列：

```

{
  "_source": {
    "include": [ "obj1.*", "obj2.*" ],
    "exclude": [ "*.description" ]
  },
  "query" : { "term" : { "type" : "syslog" } }
}

```

5. 脚本支持

对搜索是支持脚本的, 例如, 请求: `POST http://127.0.0.1:9200/secisland/log/_search?pretty`

参数如下:

```
{
  "query": {
    "term": { "type": "syslog" }
  },
  "script_fields": {
    "test1": { "script": "doc['type'].value * 2" }
  }
}
```

得到的返回值:

```
{
  "took" : 16,
  "timed_out" : false,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "secilog",
      "_type" : "log",
      "_id" : "1",
      "_score" : 1.0,
      "fields" : { "test1" : [ "syslogsyslog" ] }
    } ]
  }
}
```

4.1.2 重新评分

在 Elasticsearch 中, 搜索单个单词是比较快的, 当搜索短语时, 效率会比较低。所以 Elasticsearch 提供了重新评分的方法来提高效率。其原理主要是当在整个索引中搜索短语消耗的资源会比较多, 但大多数时候, 人们只关注最近发生的一部分文档, 所以可以先在最近的一段文档中对短语进行重新评分, 然后再查询, 这个时候看起来效率会提高很多。

在返回搜索请求节点处理排序结果之前, 在每个分片上执行重新评分。通常情况下, 当我们用重新评分 API 来重新评分的时候, 这个过程只执行一次, 在未来有可能会被调整。当查询中有排序 (sort) 或者 search_type 的设置为 scan、count 的时候将不执行重新评分。在用分页查询的时候, 每一页查询的 window_size 参数尽量不要修改, 如果修改了可能会

引起排序的混乱，导致不可预知的结果。查询重评分只会在 query 和 post_filter 阶段返回的前 K 条结果上执行二次查询，在每个分片返回文档的数量由 window_size 控制，默认返回 from 到 size 之间的个数。

默认情况下对每个文档最终的得分 (_score) 是原始得分和重新评分后的得分进行线性组合后的结果。原始评分和重新评分的比例关系分布由 query_weight 和 rescore_query_weight 控制，默认都是 1。例如重新评分查询如下。

请求：POST 127.0.0.1:9200/_search

参数：

```
{
  "query" : {
    "match" : {
      "field1" : {
        "operator" : "or",
        "query" : "the quick brown",
        "type" : "boolean"
      }
    }
  },
  "rescore" : {
    "window_size" : 50,
    "query" : {
      "rescore_query" : {
        "match" : {
          "field1" : {
            "query" : "the quick brown",
            "type" : "phrase",
            "slop" : 2
          }
        }
      }
    },
    "query_weight" : 0.7,
    "rescore_query_weight" : 1.2
  }
}
```

分数相结合的方式可以用 score_mode 控制，score_mode 的参数如下所示：

- ☐ total——将原始分数和重新评分的分值相加，为默认方式
- ☐ multiply——将原始分数和重新评分的分值相乘，对于用函数方式重新评分的时候比较有用
- ☐ avg——将原始分数和重新评分的分值进行平均
- ☐ max——在原始分数和重新评分的分值中取最大的值
- ☐ min——在原始分数和重新评分的分值中取最小的值

系统也支持顺序执行多个重新评分查询，例如：

```

{
  "query": {
    "match": {
      "field1": {
        "operator": "or",
        "query": "the quick brown",
        "type": "boolean"
      }
    }
  },
  "rescore": [ {
    "window_size": 100,
    "query": {
      "rescore_query": {
        "match": {
          "field1": {
            "query": "the quick brown",
            "type": "phrase",
            "slop": 2
          }
        }
      },
      "query_weight": 0.7,
      "rescore_query_weight": 1.2
    }, {
      "window_size": 10,
      "query": {
        "score_mode": "multiply",
        "rescore_query": {
          "function_score": {
            "script_score": {
              "script": "log10(doc['numeric'].value + 2)"
            }
          }
        }
      }
    }
  ]
}

```

先是第一个得到评分的结果，然后在第一次评分的结果上再次评分，第二次评分参考了第一次评分的排序结果，所以它可以用在第一次重新评分结果比较多的情况下利用第二次评分得到一个较小结果文档。

4.1.3 滚动查询请求

在 Elasticsearch 中，一次查询只能得到一次独立的结果，在分页中这是很不方便的。当用 Elasticsearch 进行第 n 页查询的时候，Elasticsearch 在内部查询了从开始到 n 页的所有数据，只是在返回的时候抛弃了前面 $n-1$ 页的内容。这样对查询大量数据的时候是非常不方便

的。但 Elasticsearch 提供了滚动 API 来解决此问题，这有点像数据库中的游标。

1. 滚动查询请求的原理

滚动不适合实时用户请求，适合处理比较多的数据，例如为了重建一个索引到一个新的索引中。

官方客户端只支持 Perl 和 Python 编程。



注意 从 scroll 请求返回的结果反映了在查询发生时刻的索引状态，就像一个快照。后续对文档的改动（索引、更新或者删除）都只会影响后面的搜索请求。

为了使用 scroll，在第一次搜索请求的查询中指定 scroll 参数，它告诉 Elasticsearch 需要保持搜索的上下文多长时间。例如：

请求：POST http://127.0.0.1:9200/secilog/log/_search?scroll=1m&pretty

参数：

```
{
  "query": {
    "term": { "message": "text" }
  }
}
```

保持时间的参数如下所示：

- ☐ y——年
- ☐ M——月
- ☐ w——周
- ☐ d——天
- ☐ h——小时
- ☐ m——分钟
- ☐ s——秒

返回值：

```
{
  "_scroll_id": "cXVlcnlBbmRGZXRjaDsxOzExMj01azRqYldqOVJmYTdlVW5kS94X2FnOzA7",
  "took": 1,
  "timed_out": false,
  "_shards": { "total": 1, "successful": 1, "failed": 0 },
  "hits": {
    "total": 1,
    "max_score": 0.4232868,
    "hits": [ {
      "_index": "secilog",
      "_type": "log",
```

```

    "_id" : "10",
    "_score" : 0.4232868,
    "_source" : {
      "type" : "file",
      "message" : "secilog is a log real-time analyse software,it's full
text search is based on Elasticsearch "
    }
  }
}
}

```

从返回值中可以看出，上面的请求结果中包含一个 `scroll_id`，可将这个 ID 传递给 `scroll` API 来搜索下一个批次的内容。在下一查询中的实例如下。

请求：POST http://127.0.0.1:9200/_search/scroll?pretty

参数：

```

{
  "scroll" : "1m",
  "scroll_id" : "cXVlcnlBbmRGZXRxjaDsxoZExMjolaZRqYldqOVJmYtIdlVWSk94X2FnOzA7"
}

```


返回的结果：

```

{
  "_scroll_id": "cXVlcnlBbmRGZXRxjaDsxoZExMjolaZRqYldqOVJmYtIdlVWSk94X2FnOzA7",
  "took" : 1,
  "timed_out" : false,
  "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 },
  "hits" : { "total" : 1, "max_score" : 0.4232868, "hits" : [ ] }
}

```

每一次调用 `scroll` 查询返回下一批的结果，直到返回为空 (`"hits": []`)，表示查询完成。

 **注意** 第一次搜索请求和每个后续的滚动请求返回一个新的 `_scroll_id`，只有最新的 `_scroll_id` 才能被使用。

如果请求指定了聚合 (aggregation)，只有第一次搜索的返回结果才会包含聚合结果。

当使用 `_doc` 排序查询的时候，滚动的效率是最高的，当你想遍历所有的文档且无顺序要求时，这是效率最高的方式。

请求：POST http://127.0.0.1:9200/_search?scroll=1m

参数：

```

{
  "sort": ["_doc"]
}

```

2. 搜索的时效性

scroll 参数告诉 Elasticsearch 应保持多长时间的搜索。这个值并不需要长到可以处理所有的数据，只要够处理前一批次结果的时间。每一个滚动请求设置一个新的有效时间。

一般来说，合并过程优化的背景是通过合并小段来创建一个新的大段来优化索引，然后删除小段。在执行滚动期间仍可以进行索引优化，但在打开搜索上下文而且小段仍然在使用的时候会阻止小段的删除。这就是为什么后来对文档的改变不会影响滚动搜索请求的结果。



提示 保持较旧的段，意味着需要更多的文件句柄。确保在节点中已配置访问文件句柄的参数足够用。

可以通过 stats API 检查打开了多少搜索上下文。

请求：GET http://127.0.0.1:9200/_nodes/stats/indices/search?pretty

当滚动超时的时候会删除搜索上下文，然而保持滚动打开会产生成本，所以当 scroll 不再被使用的时候需要用 clear-scroll 显式地清除滚动接口。方法如下。

请求：DELETE http://127.0.0.1:9200/_search/scroll

参数：

```
{
  "scroll_id" : ["cXVlcnlBbmRGZXRjaDsxOzExMzolzRqYldqOVJmYTdIdlVWSk94X2FnOzA7"]
}
```

可以同时清除多个 id，如下所示：

```
{
  "scroll_id" :
  ["cXVlcnlBbmRGZXRjaDsxOzExMzolzRqYldqOVJmYTdIdlVWSk94X2FnOzA7",
   "qYldqOVJmYTdIdl"]
}
```

或者使用 _all 参数清除所有 id：

DELETE http://127.0.0.1:9200/_search/scroll/_all

4.1.4 隐藏内容查询

在 Elasticsearch 中有嵌套结构和父子结构，在这两种结构中允许通过不同的范围查询搜索到不同的文档。在父 / 子情况下，可以通过查询子文档的内容返回父文档或者通过父文档查询返回子文档；在嵌套文档情况下，可以通过嵌套内部的对象查询得到嵌套文档。

在这两种情况下，返回实际匹配的内容是隐藏的。但在有些情况下，了解实际匹配的内容是非常有用的，这时就要用到隐藏内容 (inner hits) 查询。

请求：PUT <http://127.0.0.1:9200/secilog>

```
{
```



```

"mappings": {
  "weblog": {
    "properties": {
      "message": {"type": "string", "index": "analyzed"},
      "apache": {
        "type": "nested",
        "properties": {
          "method": {"type": "string", "index": "not_analyzed"},
          "status": {"type": "string", "index": "not_analyzed"}
        }
      }
    }
  }
}

```

然后插入一条记录:

请求: POST http://127.0.0.1:9200/secilog/weblog/1

参数:

```

{
  "message": "58.240.83.67 - - [16/Feb/2016:12:58:47+0800] \"POST /copyright/
check HTTP/1.1\" 200 2",
  "apache": [{"method": "POST", "status": "200"}]
}

```

我们通过嵌套查询来查询里面的文档:

请求: POST http://127.0.0.1:9200/secilog/weblog/_search/

参数:

```

{
  "query": {
    "nested": {
      "path": "apache",
      "query": {
        "bool": {
          "must": [{"term": {"apache.method": "POST"}}]
        }
      }
    }
  }
}

```

返回值:

```

{
  "took": 11,
  "timed_out": false,
  "_shards": {"total": 5, "successful": 5, "failed": 0},
  "hits": {

```

```

    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "secilog",
        "_type": "weblog",
        "_id": "1",
        "_score": 1,
        "_source": {
          "message": "58.240.83.67 - - [16/Feb/2016:12:58:47 +0800] \"POST /
            copyright/check HTTP/1.1\" 200 2",
          "apache": [{"method": "POST", "status": "200"}]}
      ]
    ]
  }
}

```

当我们加了参数 `inner_hits` 请求后:

```

{
  "query": {
    "nested": {
      "path": "apache",
      "query": {
        "bool": {
          "must": [{"term": {"apache.method": "POST"}}]}
        },
      "inner_hits": {"from": 0, "size": 10}
    }
  }
}

```

返回值:

```

{
  "took": 13,
  "timed_out": false,
  "_shards": {"total": 5, "successful": 5, "failed": 0},
  "hits": {
    "total": 1,
    "max_score": 1,
    "hits": [
      {
        "_index": "secilog",
        "_type": "weblog",
        "_id": "1",
        "_score": 1,
        "_source": {
          "message": "58.240.83.67 - - [16/Feb/2016:12:58:47 +0800] \"POST /
            copyright/check HTTP/1.1\" 200 2",
          "apache": [{"method": "POST", "status": "200"}]}
      ]
    }
  }
}

```

```

    },
    "inner_hits": {
      "apache": {
        "hits": {
          "total": 1,
          "max_score": 1,
          "hits": [
            {
              "_index": "secilog",
              "_type": "weblog",
              "_id": "1",
              "_nested": { "field": "apache", "offset": 0 },
              "_score": 1,
              "_source": { "method": "POST", "status": "200" }
            }
          ]
        }
      }
    }
  ]
}

```

从返回结果中可以看出，之前隐藏的信息都显示出来了。`_nested` 关键字在上面的例子中是至关重要的，因为它定义了内部嵌套的对象查询从哪里获取。

`inner_hits` 支持的参数如下所示：

- ☐ `from`——搜索命中的开始位置。
- ☐ `size`——搜索的文档的条数。
- ☐ `sort`——排序字段。
- ☐ `name`——在响应中用于特殊内部命中定义的名称，在一个搜索中定义了多个内部隐藏查询（inner hits）的时候比较有用。

4.1.5 搜索相关函数

Elasticsearch 提供了很多实用的函数来更加方便地满足不同的场景需求。

1. Preference

Preference 是搜索分片副本执行偏好设置。默认情况下，使用随机副本方法搜索副本的分片。可以通过 `preference` 参数设置搜索分片的范围。例如：

请求：POST `http://127.0.0.1:9200/secilog/_search?_primary`

参数：

```

{
  "query": {
    "match": {

```

```

      "message": "POST "
    }
  }
}

```

这个值也可以是一个自定义的值，它用来保证同样的分片将使用相同的自定义值，这个值可以是网络会话标识，或用户名称。例如：请求：POST http://127.0.0.1:9200/secilog/_search?preference=xyzabc123。

版本号 version：通过 "version": true 可以使每个返回的结果之中有版本号。

```

{
  "version": true,
  "query" : { "term" : { "message": "POST" } }
}

```

2. 索引加权 index_boost

当搜索一个以上的索引时，可以对每个索引配置不同的索引加权级别。当有多个索引存储类似的文档时，索引加权特别有用，使得一个索引的命中级别高于另一个索引。例如：

```

{
  "indices_boost" : {
    "index1" : 1.4,
    "index2" : 1.3
  }
}

```

3. 最小分值 min_score

可以指定搜索时候的最小评分分值。例如：

```

{
  "min_score": 0.5,
  "query" : { "term" : { "message": "POST" } }
}

```

4. 分值解释 explain

可以使每个命中的查询解释它的得分是如何计算出来的：

```

{
  "explain": true,
  "query": {
    "term": {
      "message": "copyright"
    }
  }
}

```

5. 分片情况查询

通过 `_search_shards` 可以查询分片情况，例如：

请求: GET http://127.0.0.1:9200/secilog/_search_shards/

6. 总数查询

通过 `_count` 可以查询总数, 例如:

请求: POST http://127.0.0.1:9200/secilog/_count/

参数:

```
{
  "explain": true,
  "query": {
    "term": {"message": "copyright"}
  }
}
```

返回值:

```
{
  "count": 1,
  "_shards": {"total": 5, "successful": 5, "failed": 0}
}
```

7. 是否存在查询

当设置 `size` 为 0 和 `terminate_after` 为 1 的时候可以验证查询是否有结果存在, 例如:

请求: POST http://127.0.0.1:9200/secisland/_search/

参数:

```
{
  "query": {
    "term": {"state": "close"}
  },
  "size": 0,
  "terminate_after": 1
}
```

返回值:

```
{
  "took": 2,
  "timed_out": false,
  "terminated_early": true,
  "_shards": {"total": 5, "successful": 5, "failed": 0},
  "hits": {"total": 2, "max_score": 0, "hits": []}
}
```

8. 验证接口

验证一个查询的语法是否正确, 这样可以不用实际执行, 提高效率, 防止误操作。例如:

请求: POST http://127.0.0.1:9200/secisland/_validate/query/

参数:


```
{
  "query": {
    "term": {"state": "close"}
  }
}
```

返回值:

```
{
  "valid": true,
  "_shards": {"total": 1, "successful": 1, "failed": 0}
}
```

9. 字段状态查询

通过 `_field_stats` 参数可以对索引字段的状态进行查询, 例如:

请求: GET http://127.0.0.1:9200/secisland/_field_stats?field=state/

返回值:

```
{
  "shards": {"total": 5, "successful": 5, "failed": 0},
  "indices": {
    "_all": {
      "fields": {
        "state": {
          "max_doc": 5,
          "doc_count": 5,
          "density": 100,
          "sum_doc_freq": 5,
          "sum_total_term_freq": 5,
          "min_value": "close",
          "max_value": "open"
        }
      }
    }
  }
}
```

4.1.6 搜索模板

Elasticsearch 支持基于模板的查询。可以使用 Mustache 语言作为搜索请求的预处理, 它提供了模板, 然后通过键值对来替换模板中的变量。对模板的使用有三种方式:

- ☐ 直接在请求体中使用脚本。
- ☐ 把脚本存储在索引中, 通过引用脚本 id 来使用。
- ☐ 把脚本存储在本地磁盘中, 默认的位置为: `elasticsearch\config\scripts`, 通过引用脚本名称进行使用。

下面通过举例来说明使用脚本的三种方法。

首先建一个索引, 添加一条数据。

请求: PUT http://127.0.0.1:9200/secisland/secilog/1

```
{
  "eventCount":1,
  "eventName": "linux login event"
}
```

第一种方式:

请求: POST http://127.0.0.1:9200/secisland/_search/

```
{
  "query": {
    "template": {
      "inline": {
        "match": {"eventName": "{{query_string}}"}
      },
      "params": {"query_string": "linux"}
    }
  }
}
```

上面的操作等同于:

请求: POST http://127.0.0.1:9200/secisland/_search/

```
{
  "query": {
    "match": {"eventName": "linux"}
  }
}
```

第二种方式,在用这种方式前,先要把脚本存储在 Elasticsearch 中:

请求: POST http://127.0.0.1:9200/_search/template/templateTest

```
{
  "template": {
    "query": {
      "match": {"eventName": "{{query_string}}"}
    }
  }
}
```

存储后,同时也只是 GET (查询)模板和 DELETE (删除)模板。然后通过脚本 Id 进行文档操作:

请求: POST http://127.0.0.1:9200/secisland/_search/

```
{
  "query": {
    "template": {
      "id": "templateTest",
      "params": {"query_string": "linux"}
    }
  }
}
```

第三种方式，在操作前，先要把脚本存储在文件中，文件名为 `template_test.mustache`，文件中的内容为：

```
{
  "template": {
    "query": {
      "match": {"eventName": "{{query_string}}" }
    }
  }
}
```

注意，Elasticsearch 对文件读取有个时间，刚建好后，不能生效，做验证的时候可以重启进行生效。然后在查询中使用：

请求：POST `http://127.0.0.1:9200/secisland/_search/`

```
{
  "query": {
    "template": {
      "id": "template_test",
      "params": {"query_string": "linux"}
    }
  }
}
```

系统同时支持模板验证，方法如下：

请求：POST `http://127.0.0.1:9200/_render/template`

```
{
  "inline": { // 验证请求
    "query": {
      "match": {"{{my_field}}": "{{my_value}}"}
    },
    "size": "{{my_size}}"
  },
  "params": { // 验证参数
    "my_field": "foo",
    "my_value": "bar",
    "my_size": 5
  }
}
```

返回值：

```
{
  "template_output": {
    "query": {
      "match": {"foo": "bar"} // 查询的参数替换
    },
    "size": "5"
  }
}
```

4.2 查询 DSL

本节主要介绍 Elasticsearch 查询 DSL (Domain-specific Language) 的相关知识点, DSL 查询在 Elasticsearch 中是最基本也是最重要的功能。首先介绍在 Elasticsearch 中查询和过滤的区别, 然后介绍不同类型查询的方法, 最后介绍在查询中如何进行高亮显示。

4.2.1 查询和过滤的区别

Elasticsearch 提供了基于 JSON 的完整查询 DSL 来定义查询。查询 DSL 包括两种子句:

- **叶查询子句**: 在特定的字段上查找特定的值, 比如 match、term 或者 range 查询。这些查询可以自己使用。
- **复合查询子句**: 包含其他叶查询或复合查询子句, 以合理的方式结合多条查询 (比如 bool 或 dis_max 查询), 或者改变查询行为 (比如 not 或 constant_score 查询)。
- **查询 (query)** 用于检查内容与条件是否匹配, 并且计算 _score 元字段表示匹配度。查询的结构中以 query 参数开始来执行内容查询。
- **过滤 (filter)** 不计算匹配得分, 只是简单地决定文档是否匹配。内容过滤主要用于过滤结构化的数据, 例如: 时间段是否在 2015 到 2016 年之间, status 字段是否设置为 “published”。

使用过滤往往会被 Elasticsearch 自动缓存来提高性能。

查询的子句也可以传递 filter 参数, 比如 bool 查询内的 filter、constant_score 查询内的 filter 参数等。

举个查询子句的例子。查询会匹配符合下列所有条件的文档:

- title 字段包含单词 Search。
- content 字段包含单词 Elasticsearch。
- status 字段包含准确的单词 published。
- publish_date 字段包含从 2015 年 1 月 1 日之后的日期。

请求: POST http://127.0.0.1:9200/secisland/_search

```
{
  "query": {
    "bool": {
      "must": [
        { "match": { "title": "Search" } },
        { "match": { "content": "Elasticsearch" } }
      ],
      "filter": [
        { "term": { "status": "published" } },
        { "range": { "publish_date": { "gte": "2015-01-01" } } }
      ]
    }
  }
}
```

query 参数表示内容查询。内容查询中使用的 bool 和 match 子句，用于计算每个文档的匹配得分。

filter 参数表示内容过滤。内容过滤中使用的 term 和 range 子句，会过滤掉不匹配的文档，并且不影响计算文档匹配得分。

最简单的查询，匹配所有文档，文档的 _score 值是 1.0。

```
{ "match_all": { } }
```

可以使用 boost 参数修改 _score 值：

```
{ "match_all": { "boost" : 1.2 } }
```

4.2.2 全文搜索

在文本字段上执行全文搜索。在执行查询之前，要了解被查询字段的分词方式，在查询字符串上应用每个被查询字段的映射分词器（或搜索分词器）。

标准查询接受文本 / 数字 / 日期的查询，分析参数并组成查询条件。例如：

```
{
  "match" : { "message" : "this is a test" }
}
```

注意，message 是字段名，可以用任何字段的名（包括 _all）来替换。

有三种类型的 match 查询：布尔（boolean）、短语（phrase）和短语前缀（phrase_prefix）。除此之外还有多段查询、Lucene 语法查询、简化查询。下面分别介绍。

1. 布尔查询

默认的标准查询类型，分析文本并且组成一个布尔查询。operator 参数可以设置为 or 或者 and 来控制布尔子句（默认为 or）。用于匹配的 should 子句（可选）的最小数量可以使用 minimum_should_match 参数来设置。

可以设置 analyzer 来控制在文本上执行分析过程的分词器。默认是字段映射中明确定义或者默认的搜索分词器。

lenient 参数可以设置为 true 来忽略数据类型匹配出错造成的异常，例如尝试通过文本查询字符串来查询数字类型字段。默认为 false。

（1）模糊匹配

fuzziness 可以对请求的字段类型进行模糊匹配。

prefix_length 和 max_expansions 在这种情况下可以用来控制模糊过程。如果设置模糊选项，查询会使用 top_terms_blended_freqs_\${max_expansions} 作为重写方法。fuzzy_rewrite 参数可以控制查询会怎样进行重写。

提供额外参数的例子：


```
{
  "match" : {
    "message" : {"query" : "this is a test", "operator" : "and"}
  }
}
```

(2) 零索引词查询

如果查询使用的分词器移除所有词元，默认行为是不匹配任何文档。可以使用 `zero_terms_query` 选项进行修改，接受 `none`（默认）和 `all`（相当于 `match_all` 查询）。

```
{
  "match" : {
    "message" : {
      "query" : "to be or not to be",
      "operator" : "and",
      "zero_terms_query" : "all"
    }
  }
}
```

2. 短语查询

短语查询分析文本并且创建短语查询。例如：

```
{
  "match_phrase" : {
    "message" : "this is a test"
  }
}
```

因为短语查询只是标准查询的一个类型，所以也可以用下面的方式使用：

```
{
  "match" : {
    "message" : {"query" : "this is a test", "type" : "phrase"}
  }
}
```

短语查询根据一个可配置的 `slop`（默认为 0）匹配索引词。

可以设置 `analyzer` 来控制将要在文本上执行分词的分词器。默认是字段映射中定义的分析器，或者是默认的分析器，例如：

```
{
  "match_phrase" : {
    "message" : {"query" : "this is a test", "analyzer" : "my_analyzer"}
  }
}
```

3. 短语前缀查询

可以对文本最后一个字段进行前缀匹配。例如：

```
{
  "match_phrase_prefix" : { "message" : "this is a test" }
}
```

和短语查询一样，也可以用下面的方式：

```
{
  "match" : {
    "message" : { "query" : "this is a test", "type" : "phrase_prefix" }
  }
}
```

接受短语查询相同的参数。此外，也接受 `max_expansions` 参数，可以控制最后索引词会扩展多少前缀。推荐设置为一个可以接受的值来控制查询的执行时间。例如：

```
{
  "match_phrase_prefix" : {
    "message" : { "query" : "this is a test", "max_expansions" : 10 }
  }
}
```

4. 多字段查询

在标准查询的基础上，支持多字段查询：

```
{
  "multi_match" : {
    "query": "this is a test",
    "fields": [ "subject", "message" ]
  }
}
```

字段可以通过通配符指定：

```
{
  "multi_match" : {
    "query": "Will Smith",
    "fields": [ "title", "*_name" ]
  }
}
```

个别字段可以用 caret (^) 符号加权：

```
{
  "multi_match" : {
    "query" : "this is a test",
    "fields" : [ "subject^3", "message" ]
  }
}
```

上述代码的含义是：subject 字段比 message 字段重要三倍。

多匹配查询内部执行方式取决于 type 参数，可以设置的值如下所示：

❑ `best_fields`——（默认）查找匹配任何字段的文档，但是使用最佳匹配字段的 `_score`。

- **most_fields**——查找匹配任何字段的文档，结合每个字段的 **_score**。
- **cross_fields**——用相同的分析器处理字段，把这些字段当作一个大字段。查找任何字段的每个单词。
- **phrase**——在每个字段上运行短语匹配查询，结合每个字段的 **_score**。
- **phrase_prefix**——在每个字段上运行短语前缀匹配查询，结合每个字段的 **_score**。

下面介绍这几个值。

(1) best_fields

在同一个字段中搜索多个单词的时候此参数最有用。例如，一个字段中包含“brown fox”比包含“brown”或包含“fox”更有意义。**best_fields** 类型对每个字段生成一个匹配查询并且封装成 **dis_max** 查询，来找到最佳匹配字段。例如，查询语句：

```
{
  "multi_match" : {
    "query": "brown fox",
    "type": "best_fields",
    "fields": [ "subject", "message" ],
    "tie_breaker": 0.3
  }
}
```

会执行为：

```
{
  "dis_max": {
    "queries": [
      { "match": { "subject": "brown fox" } },
      { "match": { "message": "brown fox" } }
    ],
    "tie_breaker": 0.3
  }
}
```

通常，**best_fields** 类型使用最佳匹配字段的得分，但是如果指定了 **tie_breaker**，可以通过“最佳匹配字段的得分”来计算匹配得分。

所有其他匹配字段的参数包括 **tie_breaker * _score** 相加。同时，接受 **analyzer**、**boost**、**operator**、**minimum_should_match**、**fuzziness**、**prefix_length**、**max_expansions**、**rewrite**、**zero_terms_query** 和 **cutoff_frequency**。

其中，**minimum_should_match** 参数可能的值见表 4-2。

(2) most_fields

当查询多字段包含相同文本以不同方式分词的时候此参数最有用。例如，主字段包含同义词、词根和不区分大小写的索引词；第二个字段可能包含原始的索引词；第三个字段可以包含单个索引词。通过结合所有三个字段的得分可以通过主字段匹配尽可能多的文档，通过第二个和第三个字段把最相近的结果推到列表的顶端。

表 4-2 最低匹配值

值	举例
整数	例如 3, 表示一个固定值, 与可选子句的数量无关
负整数	例如 -3, 表示可选子句的总数减去这个数字
百分比	例如 75%, 表示可选子句中必须的数量所占总数的百分比。数字向下取整
负百分比	例如 -25%, 表示可选子句中可以忽略的数量占总数的百分比。数字向下取整, 再被从总数中减去来确定最小值
结合	例如 3<90%, 表示如果可选子句的数量小于整数部分, 则这些子句都必须匹配, 如果大于整数部分, 这些子句总数的百分比部分是必须的

查询语句:

```
{
  "multi_match" : {
    "query": "quick brown fox",
    "type": "most_fields",
    "fields": [ "title", "title.original", "title.shingles" ]
  }
}
```

会执行为:

```
{
  "bool": {
    "should": [
      { "match": { "title": "quick brown fox" } },
      { "match": { "title.original": "quick brown fox" } },
      { "match": { "title.shingles": "quick brown fox" } }
    ]
  }
}
```

最终的匹配得分是将每个匹配子句的得分加在一块, 然后除以匹配子句的数量。

接受的参数有: analyzer、boost、operator、minimum_should_match、fuzziness、prefix_length、max_expansions、rewrite、zero_terms_query 和 cutoff_frequency。

(3) cross_fields

当结构化的文档中多个字段应该匹配的时候, 此参数特别有用。例如, 当通过 first_name 和 last_name 字段查询 “Will Smith” 的时候, 最佳的匹配是 “Will” 在一个字段, “Smith” 在另一个字段。

一种处理这种查询的简单方式是将 first_name 和 last_name 字段索引到一个 full_name 字段。当然, 这只能在索引的时候完成。

cross_fields 类型在查询时通过采取 term-centric 方法来尝试解决这个问题。首先将查询字符串分词为单独的索引词, 然后在任一字段中查找每个索引词。

查询示例如下:

```
{
  "multi_match" : {
    "query": "Will Smith",
    "type": "cross_fields",
    "fields": [ "first_name", "last_name" ],
    "operator": "and"
  }
}
```

接受的参数有: analyzer、boost、operator、minimum_should_match、zero_terms_query 和 cutoff_frequency。

(4) phrase 和 phrase_prefix

短语和短语前缀类型和 best_fields 一样,只不过使用的是 match_phrase 查询或者 match_phrase_prefix 查询而不是 match 查询。

查询示例如下:

```
{
  "multi_match" : {
    "query": "quick brown f",
    "type": "phrase_prefix",
    "fields": [ "subject", "message" ]
  }
}
```

会执行为:

```
{
  "dis_max": {
    "queries": [
      { "match_phrase_prefix": { "subject": "quick brown f" } },
      { "match_phrase_prefix": { "message": "quick brown f" } }
    ]
  }
}
```

接受如下参数: analyzer、boost、slop 和 zero_terms_query 参数。此外,短语前缀类型接受 max_expansions。

(5) cross_field 和分词器

cross_field 类型只能够工作在使用相同分词器的 term-centric 模式的字段。具有相同分词器的字段被分在一组,如果有多个分组,会通过 bool 查询结合在一起。

例如,first 和 last 字段拥有相同的分词器,添加 first.edge 和 last.edge 使用 edge_ngram 分析器,查询:

```
{
  "multi_match" : {
    "query": "Jon",
    "type": "cross_fields",

```



```

    "fields": ["first", "first.edge", "last", "last.edge"]
  }
}

```

first 和 last 会被组在一起作为单独的字段, first.edge 和 last.edge 会被组在一起作为单独的字段。

可以重写查询作为两个通过 bool 查询联合在一起的 cross_fields 查询, 然后对其中一个应用 minimum_should_match 参数:

```

{
  "bool": {
    "should": [
      {
        "multi_match": {
          "query": "Will Smith",
          "type": "cross_fields",
          "fields": ["first", "last"],
          "minimum_should_match": "50%"
        }
      },
      {
        "multi_match": {
          "query": "Will Smith",
          "type": "cross_fields",
          "fields": ["*.edge"]
        }
      }
    ]
  }
}

```

也可以强迫所有字段到相同的分组, 方法是在查询中指定 analyzer 参数:

```

{
  "multi_match": {
    "query": "Jon",
    "type": "cross_fields",
    "analyzer": "standard",
    "fields": ["first", "last", "*.edge"]
  }
}

```

5. Lucene 语法查询

通过使用语法分析器来分析内容进行查询:

```

{
  "query_string": {
    "default_field": "content",
    "query": "this AND that OR thus"
  }
}

```

query_string 为最高级别，参数见表 4-3。

表 4-3 查询字符串参数

参数	含义
query	实际被解析的查询
default_field	如果没有指定前缀字段，查询索引词的默认字段。默认为 index.query.default_field 索引设置，这项设置的默认值为 _all
default_operator	如果没有明确指定操作符，使用默认操作符（OR）。例如，利用默认操作符 OR 查询字符串 capital of Hungary，翻译为 capital OR of OR Hungary
analyzer	用来分析查询字符串的分词器名称
allow_leading_wildcard	设置 * 或 ? 能否作为第一个字符。默认值为 true
lowercase_expanded_terms	通配符、前缀、模糊和范围索引词是否被自动转换为小写字母（索引词不被分词）。默认值为 true
fuzzy_max_expansions	控制模糊查询将扩展的索引词数量。默认值为 50
fuzziness	设置模糊查询的模糊性。默认值为 AUTO
fuzzy_prefix_length	设置模糊查询的前缀长度，默认值为 0
phrase_slop	设置短语的默认溢出。如果是零，表示精确的短语匹配。默认值为 0
boost	设置查询的加权值。默认值为 1.0
analyze_wildcard	默认的，通配符索引词在查询字符串中不会被分词。通过设置为 true，会尽可能的进行分词
auto_generate_phrase_queries	默认值为 false
max_determinized_states	限制多少正则表达式允许被创建。防止太困难的正则表达式。默认值为 10 000
minimum_should_match	控制多少“should”子句在布尔查询的结果中应该被匹配。可以是一个确切值（例如，2）、一个百分比（例如，20%）或者两种形式结合起来
lenient	如果设置为 true，会忽略基于格式的失败（例如，向数字型字段提供文本）
locale	用于字符串转换的区域，默认值为 ROOT
time_zone	应用于任何基于日期范围查询的时间区间

（1）默认字段

如果在查询字符串语法中没有明确指出查询的字段，会 index.query.default_field 提取需要搜索的字段。默认为 _all 字段。

所以，如果 _all 字段被禁用，将其修改设置为一个不同的默认字段是很有必要的。

（2）多字段

query_string 查询也可以在多个字段上执行。可以通过“fields”参数定义字段。

多字段执行 query_string 查询的意义在于利用 OR 子句扩展每个查询索引词，形如：

```
field1:query_term OR field2:query_term | ...
```

例如，下面的查询：

```
{
```

```
"query_string" : {"fields" : ["content", "name"], "query" : "this AND that"}
```

实际匹配形式:

```
{
  "query_string": {
    "query": "(content:this OR name:this) AND (content:that OR name:that)"
  }
}
```

(3) 语法

查询字符串被解析为一系列字段和运算符。字段可以是一个单词或者短语(搜索短语中的所有单词,以相同的顺序用双引号包起来)。

可用的选项有:

1) 字段名——可以指定 `default_field` 之外的其他字段。

☐ `status` 字段包含 `active`。

```
status:active
```

☐ `title` 字段包含 `quick` 或 `brown`。

```
title:(quick OR brown)
```

```
title:(quick brown)
```

☐ `author` 字段包含短语 “John Smith”。

```
author:"John Smith"
```

☐ 任何 `book.title`、`book.content` 或 `book.date` 字段包含 `quick` 或 `brown` (注意,需要用反斜线转义 `*`)。

```
book.\*:(quick brown)
```

☐ `title` 字段不包含(或者缺失)值。

```
_missing_:title
```

☐ `title` 字段包含任何非空值。

```
_exists_:title
```

2) 通配符——通配符搜索使用 `?` 代替单个字符,使用 `*` 代替零个或多个字符。

```
qu?ck bro*
```

3) 正则表达式——通过使用斜线(`/`)包围,正则表达式可以植入到查询字符串中。

```
name:/joh?n(ath[oa]n)/
```

4) 模糊化。

```
quikc~1
```

5) 邻近搜索——可以指定短语中单词的最大编辑距离。

```
"fox quick"~5
```

字段中的文本越接近于查询字符串的顺序，文档的相关性越高。短语“quick fox”的相关性高于“quick brown fox”。

6) 范围——可以指定日期、数字或字符串字段的范围。包含的范围用方括号指定 [min TO max]，不包含的范围用花括号指定 {min TO max}。

□ 2016 年的所有日期

```
date:[2016-01-01 TO 2016-12-31]
```

□ 数字 1 到 5

```
count:[1 TO 5]
```

□ alpha 和 omega 之间的标签，不包括 alpha 和 omega

```
tag:{alpha TO omega}
```

□ 大于等于 10 的数

```
count:[10 TO *]
```

□ 2016 年之前的日期

```
date:{* TO 2016-01-01}
```

□ 大于等于 1 小于 5 的数

```
count:[1 TO 5)
```

□ 一边值为无限

```
age:>10
```

```
age:>=10
```

```
age:<10
```

```
age:<=10
```

7) 增权。

```
quick^2 fox
```

```
"john smith"^2
```

```
(foo bar)^4
```

8) 布尔操作符。

+ 必须包含。

- 必须不包含。

支持 AND、OR 和 NOT (也可以写作 &&、|| 和 !) 操作符。

```
((quick AND fox) OR (brown AND fox) OR fox) AND NOT news
```

等同于：

```
{
  "bool": {
```

```

    "must": { "match": "fox" },
    "should": { "match": "quick brown" },
    "must_not": { "match": "news" }
  }
}

```

9) 分组——多字段或子句可以使用圆括号进行分组。

```
status:(active OR pending) title:(full text search)^2
```

10) 保留字符——查询中出现保留字符的时候, 需要进行转义。

```
\(1+1\)\\=2
```

保留字符有: + - = && || > < ! () { } [] ^ " ~ * ? : \ /

11) 空查询——如果查询字符串是空的或者仅包含空格, 查询会生成一个空结果集。

6. 简化查询

系统提供了简化语法来进行查询。不像普通的 Lucene 语法查询, 简化查询不会抛出异常, 而且会丢弃查询无效部分:

```

{
  "simple_query_string" : {
    "query": "\"fried eggs\" +(eggplant | potato) -frittata",
    "analyzer": "snowball",
    "fields": ["body^5", "_all"],
    "default_operator": "and"
  }
}

```

简化查询可以接受参数见表 4-4。

表 4-4 简化查询参数

参数	说明
query	解析用于实际查询的内容
fields	用来执行查询解析的字段。默认是 index.query.default_field 索引设置, 返回的默认值为 _all
default_operator	如果没有明确指定运算符, 默认使用的运算符。默认值是 OR
analyzer	当创建混合查询时用来分析查询中每个索引词的分析器
flags	指定简化查询启用的功能。默认值为 ALL
lowercase_expanded_terms	前缀或模糊查询里的索引词是否应该被自动转换为小写。默认值为 true
analyze_wildcard	前缀查询索引词是否应该被自动分析。如果是 true, 会尽量来分析前缀。然而, 一些分析器仅基于索引词前缀不能提供一个有意义的结果。默认值为 false
locale	用于字符串转换的位置。默认值为 ROOT
lenient	如果设置为 true, 会导致基于格式的错误 (比如向数字型字段提供文本) 将被忽略
minimum_should_match	返回的文档必须匹配的最小子句数量

（1）语法

简单查询字符串支持的特殊字符如下所示：

+ 表示 AND 运算符。

| 表示 OR 运算符。

- 排除一个词元。

" 包含一批词元来指定搜索短语。

* 在索引词结尾表示前缀查询。

(和) 表示优先。

~N 在单词的后面，表示编辑距离（模糊性）。

~N 在短语的后面，表示溢出量。

为了搜索任何这些特殊字符，需要使用斜线 (\)。

（2）默认字段

当在搜索字符串语法中没有明确指出需要搜索的字段时，`index.query.default_field` 会用来提取要搜索的字段。默认为 `_all` 字段。

（3）多字段

字段参数也可以包括基于模式的字段名，可以自动扩展相关字段（动态引入包含的字段）。例如：

```
{
  "simple_query_string" : { "fields" : ["content", "name.*^5"],
    "query" : "foo bar baz" }
}
```

（4）标记

简单查询字符串支持多个标记来指定需要启用的解析功能。在 `flags` 参数中指定为一个用 | 分隔的字符串：

```
{
  "simple_query_string" : { "query" : "foo | bar + baz*",
    "flags" : "OR|AND|PREFIX" }
}
```

可用的标记有：ALL、NONE、AND、OR、NOT、PREFIX、PHRASE、PRECEDENCE、ESCAPE、WHITESPACE、FUZZY、NEAR 和 SLOP。

4.2.3 字段查询

全文本查询在执行之前会分析查询字符串，而字段查询只针对存储在反向索引中的精确索引词。

这些查询通常用于结构化数据，例如数字、日期和枚举，而不是全文本字段。或者，先前的分析过程可以用来处理低等级查询。

1. 单字段查询

根据指定字段中包含的指定内容查找文档。

```
{
  "term" : { "user" : "Kimchy" }
}
```

上代码指在反向索引中查找 user 字段包含 kimchy 的文档。

2. 多字段查询

过滤文档，文档字段匹配任何提供的索引词（不分词）。例如：

```
{
  "constant_score" : {
    "filter" : { "terms" : { "user" : ["kimchy", "elasticsearch"] } }
  }
}
```

3. 范围查询

根据指定字段包含的值（日期、数字或字符串）范围查找文档。

Lucene 查询类型取决于字段类型，对于字符串类型字段，使用 TermRangeQuery；对于数字/日期类型字段，使用 NumericRangeQuery。例如查询 age 字段值在 10 到 20 之间的文档：

```
{
  "range" : {
    "age" : { "gte" : 10, "lte" : 20, "boost" : 2.0 }
  }
}
```

范围查询接受的参数如下所示：

□ gte——大于或等于。

□ gt——大于。

□ lte——小于或等于。

□ lt——小于。

□ boost——设置查询的加权值，默认为 1.0。

(1) 日期型字段范围

当在日期型字段上运行范围查询，可以使用名为“日期匹配”的分段来指定范围：

```
{
  "range" : { "date" : { "gte" : "now-1d/d", "lt" : "now/d" } }
}
```

日期匹配和取整。当使用日期匹配来凑整日期到最近的月、日、小时，等等，凑整的日期取决于是否包含日期范围的结尾。向上舍入到舍入范围的最后毫秒数，向下舍入到舍入范

围的第一个毫秒数, 参见如下所示:

- **gt**——大于日期向上舍入: 2016-01-18||/M 成为 2016-01-31T23:59:59.999, 不包含整个月。
- **gte**——大于或等于日期向下舍入: 2016-01-18||/M 成为 2016-01-01, 包含整个月。
- **lt**——小于日期向下舍入: 2016-01-18||/M 成为 2016-01-01, 不包含整个月。
- **lte**——小于或等于日期向上舍入: 2016-01-18||/M 成为 2016-01-31T23:59:59.999, 包含整个月。

范围查询的日期格式。格式化的日期默认使用指定在日期型字段上的格式来分析, 可以通过传递 **format** 参数到范围查询中来重写日期格式:

```
{
  "range" : { "born" : { "gte": "01/01/2012", "lte": "2013",
    "format": "dd/MM/yyyy|yyyy" } }
}
```

范围查询的时间区间。通过在日期值里指定时间区间(如果格式接受时间区间)或者在 **time_zone** 参数中指定时间区间, 日期可以从其他时间区间转换到世界标准时间:

```
{
  "range" : {
    "timestamp" : { "gte": "2015-01-01 00:00:00", "lte": "now",
      "time_zone": "+01:00" }
  }
}
```

日期将被转换为 2014-12-31T23:00:00 UTC。

now 不会受 **time_zone** 参数影响(日期必须存储为 UTC)。

(2) 是否存在查询

查找指定字段包含任何非空值的文档:

```
{
  "exists" : { "field" : "user" }
}
```

例如, 匹配查询的文档:

```
{ "user": "jane" }
{ "user": "" }
{ "user": "-" }
{ "user": ["jane"] }
{ "user": ["jane", null] }
```

不匹配查询的文档:

```
{ "user": null }
{ "user": [] }
{ "user": [null] }
{ "foo": "bar" }
```

(3) 为空查询

```
"bool": {
  "must_not": {
    "exists": {"field": "user"}
  }
}
```

这个查询返回 user 字段没有值的文档。

(4) 前缀查询

匹配文档，字段包含拥有特定前缀的索引词（不分词）。前缀查询对应 Lucene 中的 PrefixQuery。例如，查询 user 字段包含以 ki 开头的索引词的文档：

```
{"prefix" : { "user" : "ki" }}
```

(5) 通配符查询

匹配文档，字段匹配通配符表达式（不分词）。通配符 * 匹配任意字符（包含空字符），通配符 ? 匹配任何单个字符。注意这个查询可能会比较缓慢，需要在许多索引词上面重复执行。为了避免极端缓慢的通配符查询，通配符索引词不应该以一个通配符开头。通配符查询对应 Lucene 中的 WildcardQuery：

```
{"wildcard" : { "user" : "ki*y" }}
```

(6) 模糊查询

模糊查询对字符串型字段使用基于编辑距离的相似性，以及数字型和日期型字段的正负 (+/-) 范围进行匹配。

字符串型字段。模糊查询基于 fuzziness 指定的最大编辑距离生成所有可能匹配的索引词，然后检查索引词字典来找出确实存在于索引中的索引词。

```
{
  "fuzzy" : {
    "user" : {
      "value" : "ki",
      "boost" : 1.0,
      "fuzziness" : 2,
      "prefix_length" : 0,
      "max_expansions": 100
    }
  }
}
```

参数如下：

□ fuzziness——最大编辑距离，默认为 AUTO。

□ prefix_length——不会被“模糊化”的最初字符的数量。可以用来减少必须审查的索引词数量，默认值为 0。

□ `max_expansions`——模糊查询将要扩展的索引词最大数量。默认是 50。

数字型和日期型字段。使用 `fuzziness` 值作为范围来执行模糊查询：

```
-fuzziness <= field value <= +fuzziness
```

例如：

```
{
  "fuzzy" : {
    "price" : { "value" : 12, "fuzziness" : 2 }
  }
}
```

会查询 `value` 值的范围在 10 到 14 之间的结果。日期字段支持时间值，例如：

```
{
  "fuzzy" : {
    "created" : { "value" : "2010-02-05T12:05:07", "fuzziness" : "1d" }
  }
}
```

(7) 类型查询

根据匹配提供的映射类型来过滤文档：

```
{
  "type" : { "value" : "secilog" }
}
```

(8) 主键查询

获取只拥有提供的主键的文档。注意，查询使用 `_uid` 字段：

```
{
  "ids" : { "type" : "secilog", "values" : ["1", "4", "100"] }
}
```

`type` 选项是可选的，也可以接受数组形式的值。如果没有指定类型，所有定义在索引映射中的类型都会被尝试。

4.2.4 复合查询

复合查询是包含其他子查询的查询，还可以再包含复合查询，并结合结果和匹配得分来改变它们的行为，或者可以从查询中进行内容过滤。

1. 常数得分查询

这个查询包含另一个查询，并且仅返回过滤查询中任何常数得分等于查询加权的文档。对应 Lucene 中的 `ConstantScoreQuery`：

```
{
  "constant_score" : {
```



```


    "filter" : { "term" : { "user" : "kimchy" } },
    "boost" : 1.2
  }
}

```

2. 布尔查询

获取匹配其他查询的布尔值的文档。布尔查询对应 Lucene 的 BooleanQuery。基于一个或多个布尔子句的使用，每个子句都有一类事件：

- ❑ **must**——必须出现在匹配文档中，并且会影响匹配得分。
- ❑ **filter**——必须出现在匹配文档中，匹配得分将会被忽略。
- ❑ **should**——应该出现在匹配文档中，在布尔查询中如果没有 **must** 或 **filter** 子句，文档必须匹配一个或多个 **should** 子句。应该匹配的 **should** 子句的最小数量可以通过 **minimum_should_match** 参数进行设置。
- ❑ **must_not**——必须不出现在匹配文档中。

 **注意** 如果查询用于过滤内容而且有 **should** 子句，那么最少一个 **should** 子句需要被匹配。

布尔查询也支持 **disable_coord** 参数（默认为 **false**）。

布尔查询采取匹配越多越好的方式，所以每个匹配的 **must** 或 **should** 子句的得分会被加在一起，为每个文档提供最终的 **_score**。

```

{
  "bool" : {
    "must" : { "term" : { "user" : "kimchy" } },
    "filter" : { "term" : { "tag" : "tech" } },
    "must_not" : {
      "range" : { "age" : { "from" : 10, "to" : 20 } }
    },
    "should" : [ { "term" : { "tag" : "wow" } },
      { "term" : { "tag" : "elasticsearch" } }
    ],
    "minimum_should_match" : 1,
    "boost" : 1.0
  }
}

```

bool.filter 匹配得分。

在 **filter** 元素下指定的查询对匹配得分没有影响——返回得分为 0，得分仅受指定查询的影响。例如获取所有 **status** 字段包含 **active** 索引词的文档：

请求：GET http://127.0.0.1:9200/secisland/_search{

```

  "query": {
    "bool": {
      "filter": {

```

```

    "term": {"status": "active"}
  }
}
}

```

这个查询对所有文档指定匹配得分为 0，因为没有指定查询获取匹配得分。

请求: GET http://127.0.0.1:9200/secisland/_search{

```

  "query": {
    "bool": {
      "query": {"match_all": {}},
      "filter": {
        "term": {
          "status": "active"
        }
      }
    }
  }
}

```

拥有 match_all 查询的布尔查询对每个文档指定匹配值为 1.0。

请求: GET http://127.0.0.1:9200/secisland/_search{

```

  "query": {
    "constant_score": {
      "filter": {
        "term": {"status": "active"}
      }
    }
  }
}

```

constant_score 查询对所有通过过滤匹配的文档指定匹配值为 1.0。

如果需要知道查询返回的文档匹配布尔查询的哪一条查询子句，可以使用命名查询对每个子句进行命名。

3. 最大值获取查询

这个查询通过执行自己的子查询生成文档的并集，并且用文档执行任何子查询的最大匹配得分作为文档得分。

查询对应 Lucene 的 DisjunctionMaxQuery。

```

{
  "dis_max" : {
    "tie_breaker" : 0.7,
    "boost" : 1.2,
    "queries" : [
      {"term" : { "age" : 34 }},
      {"term" : { "age" : 35 }}
    ]
  }
}

```

```

    }
  }
}

```

4. boosting 查询

可以用来有效降级匹配给出的查询结果。不像布尔查询的“NOT”子句，boosting 查询仍然选择包含不合需要的索引词的文档。但降低了它们的整体得分：

```

{
  "boosting" : {
    "positive" : {
      "term" : { "field1" : "value1" }
    },
    "negative" : {
      "term" : { "field2" : "value2" }
    },
    "negative_boost" : 0.2
  }
}

```

5. 指定索引查询

当搜索在多个索引中执行的时候，指定索引查询是有用的。可以指定一个索引名的列表和一个内部查询，这个内部查询仅在索引名匹配列表时执行。如果搜索其他不匹配列表的索引时，执行 no_match_query：

```

{
  "indices" : {
    "indices" : ["index1", "index2"],
    "query" : { "term" : { "tag" : "wow" } },
    "no_match_query" : { "term" : { "tag" : "kow" } }
  }
}

```

可以使用 indices 字段来提供单独的索引。no_match_query 也可以赋值为 none（不匹配任何文档），或者 all（匹配所有文档）。默认值为 all。



提示 字段顺序是重要的，indices 在 query 和 no_match_query 之前提供，相关查询只在将要执行的索引中解析。这可以用来避免查询在不需要的时候解析，也可以防止潜在的映射错误。

6. AND、OR、NOT 查询

在其他查询中使用 AND、OR、NOT 布尔操作符匹配文档。

```

{
  "filtered" : {

```

```

"query" : { "term" : { "name.first" : "shay" } },
"filter" : {
  "and" : [
    {
      "range" : {
        "postDate" : { "from" : "2010-03-01", "to" : "2010-04-01" }
      }
    },
    { "prefix" : { "name.second" : "ba" } }
  ]
}
}
}

```

7. 过滤查询

过滤查询仅用来过滤结果集，并且另一个集合用来计算匹配得分的查询。



提示 通过过滤尽可能多地排除文档，然后仅在剩下的文档中执行查询。

请求：GET http://127.0.0.1:9200/secisland/_search {

```

"query": {
  "filtered": {
    "query": { "match": { "secilog": "full text search" } },
    "filter": {
      "range": { "created": { "gte": "now-1d/d" } }
    }
  }
}
}

```

过滤查询作为 query 参数的值传递给搜索请求。

(1) 不包含查询的过滤

如果不指定查询，默认包含全匹配查询：

请求：GET http://127.0.0.1:9200/secisland/_search{

```

"query": {
  "filtered": {
    "filter": { "range": { "created": { "gte": "now-1d/d" } } }
  }
}
}

```

没有指定查询，所以这个请求仅应用过滤，返回一天内创建的所有文档。

(2) 多重过滤

多重过滤可以通过包含布尔查询来使用：

```

{
  "filtered": {
    // 父文档匹配，通过 parent_type 指定。返回关联匹配的父文档的子文

```

```

"query": { "match": { "secilog": "full text search" }},
"filter": {
  "bool": {
    "must": { "range": { "created": { "gte": "now-1d/d" } }},
    "should": [{ "term": { "featured": true }},
                { "term": { "starred": true } }],
    "must_not": { "term": { "deleted": false } }
  }
}
}

```

8. 限制查询

限制查询限制执行文档（每个分片）的数量：

```

{
  "filtered" : {
    "filter" : { "limit" : { "value" : 100 }},
    "query" : { "term" : { "name.first" : "shay" } }
  }
}

```

4.2.5 连接查询

在 Elasticsearch 这种分布式系统中执行完整的 SQL 类型的表关联查询的代价是昂贵的。但 Elasticsearch 还是提供了两种方式的关联，一种是嵌套关系，一种是父子关系，同时系统提供了这两种关联的查询。

1. 嵌套查询

嵌套查询可以查询嵌套对象 / 文档。执行的查询把嵌套对象 / 文档视作父文档的单独文档。

简单的嵌套查询用法示例：

```

{
  "nested" : {
    "path" : "obj1",
    "score_mode" : "avg",
    "query" : {
      "bool" : {
        "must" : [
          { "match" : { "obj1.name" : "blue" } },
          { "range" : { "obj1.count" : { "gt" : 5 } } }
        ]
      }
    }
  }
}

```


path 指出嵌套对象的路径, query 查询的范围是在匹配路径的嵌套文档中。注意任何在查询中引用的字段必须使用全路径。

score_mode 可以设置内部子匹配如何影响父匹配的得分。默认为 avg, 也可以设置为 sum、min、max 和 none。

自动支持和检查多级别嵌套, 如果内部嵌套查询存在于其他嵌套查询中, 会自动匹配相关嵌套级别。

2. 父/子文档查询

(1) 子文档查询

子文档查询将父文档拥有的查询返回匹配到的子文档:

```
{
  "has_child" : {
    "type" : "blog_tag",
    "query" : { "term" : { "tag" : "something" } }
  }
}
```

子文档查询也支持匹配得分。支持的模式有 min、max、sum、avg 或 none。默认值为 none, 表示产生和前版本相同的行为。如果得分模式设置为其他值, 所有匹配的子文档得分聚合成相关的父文档得分。在 has_child 查询中通过 score_mode 字段来指定得分类型:

```
{
  "has_child" : {
    "type" : "blog_tag",
    "score_mode" : "sum",
    "query" : {
      "term" : { "tag" : "something" }
    }
  }
}
```

子文档查询可以指定最小或者最大数量的子文档:

```
{
  "has_child" : {
    "type" : "blog_tag",
    "score_mode" : "sum",
    "min_children": 2,
    "max_children": 10,
    "query" : {
      "term" : { "tag" : "something" }
    }
  }
}
```

(2) 父文档查询

父文档查询在父文档空间执行, 通过 parent_type 指定。返回关联匹配的父文档的子文

档。工作方式和存在子文档查询一样，也接受相同的参数。

```
{
  "has_parent" : {
    "parent_type" : "blog",
    "query" : {
      "term" : { "tag" : "something" }
    }
  }
}
```

父文档查询支持的得分类型是 `score` 或者 `none`。默认值是 `none`，忽略从父文档得到的匹配得分。在这种情况下，匹配得分基于查询增益（默认是 1）。如果得分类型设置为 `score`，父文档的匹配得分聚合到属于它的子文档。可以使用 `score_mode` 字段来指定得分模式：

```
{
  "has_parent" : {
    "parent_type" : "blog",
    "score_mode" : "score",
    "query" : {
      "term" : { "tag" : "something" }
    }
  }
}
```

4.2.6 地理查询

Elasticsearch 支持两种地理数据类型的字段：“地理点”类型，支持经度/纬度对；“地理形状”类型，支持点、线、圈、多边形、多边形集合等。

1. 地理形状查询

使用地理形状类型过滤索引的文档。需要地理形状映射。

地理形状查询使用相同的方格表示为地理形状映射来查找文档，拥有与查询形状相交的形状。

查询支持两种方式来定义查询形状，通过提供整个形状的定义或者引用预索引在其他索引中的形状名。

(1) 形状定义

与地理形状类型相同，地理形状过滤使用 GeoJSON 来表示形状。

```
{
  "query": {
    "bool": {
      "must": {
        "match_all": {}
      },
      "filter": {
```

```

    "geo_shape": {
      "location": {
        "shape": {
          "type": "envelope",
          "coordinates": [[13.0, 53.0], [14.0, 52.0]]
        },
        "relation": "within"
      }
    }
  }
}

```

(2) 预索引形状

查询也支持使用被索引在其他索引或者索引类型中的形状。当存在一个预定义的形状列表时，可以使用逻辑名（例如南京）来引用而不用每次都提供具体的坐标值。在这种情况下，只需要提供的参数见如下所示：

- id——包含预索引形状文档 ID。
- index——预索引形状的索引名。默认为 shapes。
- type——预索引形状索引类型。
- path——指定的字段作为包含预索引形状的路径。默认为 shape。

```

{
  "bool": {
    "must": { "match_all": {} },
    "filter": {
      "geo_shape": {
        "location": {
          "indexed_shape": {
            "id": "DEU",
            "type": "countries",
            "index": "shapes",
            "path": "location"
          }
        }
      }
    }
  }
}

```

(3) 空间关系

地理形状映射参数决定搜索时使用的空间关系操作符。

所有可以使用的空间关系操作符列表如下所示：

- INTERSECTS——返回地理形状字段和查询集合相交的所有文档（默认）。
- DISJOINT——返回地理形状字段和查询集合没有关联的所有文档。

□ WITHIN——返回地理形状字段在查询集合内的所有文档。

□ CONTAINS——返回地理形状字段包含查询集合的所有文档。

2. 地理范围查询

可以基于一个位置点的范围来过滤查询文档。

```
{
  "bool" : {
    "must" : {
      "match_all" : {}
    },
    "filter" : {
      "geo_bounding_box" : {
        "pin.location" : {
          "top_left" : {"lat" : 40.73, "lon" : -74.1},
          "bottom_right" : {"lat" : 40.01, "lon" : -71.12}
        }
      }
    }
  }
}
```

查询选项如下所示：

□ _name——可选名称字段来定义过滤请求。

□ ignore_malformed——设置为 true 来接受无效的经度或纬度定义的地理点（默认是 false）。

□ type 设置为 indexed 或者 memory 中的一个，来定义过滤会在索引还是内存中执行。默认是 memory。

接受的格式如下。

经纬度作为参数：

```
"filter" : {
  "geo_bounding_box" : {
    "pin.location" : {
      "top_left" : {"lat" : 40.73, "lon" : -74.1},
      "bottom_right" : {"lat" : 40.01, "lon" : -71.12}
    }
  }
}
```

经纬度数组：

```
"filter" : {
  "geo_bounding_box" : {
    "pin.location" : {
      "top_left" : [-74.1, 40.73],
      "bottom_right" : [-71.12, 40.01]
    }
  }
}
```

经纬度字符串:

```

"filter" : {
  "geo_bounding_box" : {
    "pin.location" : {
      "top_left" : "40.73, -74.1",
      "bottom_right" : "40.01, -71.12"
    }
  }
}

```

地理散列:

```

"filter" : {
  "geo_bounding_box" : {
    "pin.location" : {
      "top_left" : "dr5r9ydj2y73",
      "bottom_right" : "drj7teegpus6"
    }
  }
}

```

地理范围的顶点可以通过 `top_left` 和 `bottom_right` 设置, 或者通过 `top_right` 和 `bottom_left` 参数设置。除了成对设置经纬度值, 可以使用 `top`、`left`、`bottom` 和 `right` 分别设置经纬度值:

```

"filter" : {
  "geo_bounding_box" : {
    "pin.location" : {
      "top" : -74.1,
      "left" : 40.73,
      "bottom" : -71.12,
      "right" : 40.01
    }
  }
}

```

3. 地理距离查询

获取一个地理点确切距离内的所有文档:

```

{
  "bool" : {
    "must" : {
      "match_all" : {}
    },
    "filter" : {
      "geo_distance" : {

```



```

    "distance" : "200km",
    "pin.location" : {
      "lat" : 40,
      "lon" : -70
    }
  }
}

```

接受的格式如下。

经纬度作为参数：

```

"filter" : {
  "geo_distance" : {
    "distance" : "12km",
    "pin.location" : {
      "lat" : 40,
      "lon" : -70
    }
  }
}

```

经纬度数组：

```

"filter" : {
  "geo_distance" : {
    "distance" : "12km",
    "pin.location" : [-70, 40]
  }
}

```

经纬度字符串：

```

"filter" : {
  "geo_distance" : {
    "distance" : "12km",
    "pin.location" : "40,-70"
  }
}

```

地理散列：

```

"filter" : {
  "geo_distance" : {
    "distance" : "12km",
    "pin.location" : "drm3btev3e86"
  }
}

```

4. 地理距离范围查询

获取存在于一个地理点一定距离范围内的所有文档：

```

{
  "bool" : {
    "must" : { "match_all" : {} },
    "filter" : {
      "geo_distance_range" : {
        "from" : "200km",
        "to" : "400km",
        "pin.location" : { "lat" : 40, "lon" : -70 }
      }
    }
  }
}

```

支持和地理距离过滤相同的位置点参数和查询选项。也支持通用的范围参数: lt、lte、gt、gte、from、to、include_upper 和 include_lower。

5. 多边形地理查询

查询可以包含命中多边形内的点:

```

{
  "bool" : {
    "query" : { "match_all" : {} },
    "filter" : {
      "geo_polygon" : {
        "person.location" : {
          "points" : [
            { "lat" : 40, "lon" : -70 },
            { "lat" : 30, "lon" : -80 },
            { "lat" : 20, "lon" : -90 }
          ]
        }
      }
    }
  }
}

```

接受的格式如下。

经纬度数组:

```

"filter" : {
  "geo_polygon" : {
    "person.location" : {
      "points" : [
        [-70, 40],
        [-80, 30],
        [-90, 20]
      ]
    }
  }
}

```

经纬度字符串:

```

"filter" : {
  "geo_polygon" : {
    "person.location" : {
      "points" : [
        "40, -70",
        "30, -80",
        "20, -90"
      ]
    }
  }
}

```

地理散列:

```

"filter" : {
  "geo_polygon" : {
    "person.location" : {
      "points" : [
        "drn5xlg8cu2y",
        "30, -80",
        "20, -90"
      ]
    }
  }
}

```

6. 地理散列单元查询

地理散列单元查询提供地理散列层次结构的访问。通过定义地理散列单元，只有单元里的地理点会匹配这个查询。

为了使过滤生效，所有地理散列前缀需要添加到索引中。例如一个地理散列 u30 需要分解为三个索引词：u30、u3 和 u。这个分解必须在需要执行过滤的地理字段映射中启用，通过设置 `geohash_prefix` 选项：

```

{
  "mappings" : {
    "location": {
      "properties": {
        "pin": {
          "type": "geo_point",
          "geohash": true,
          "geohash_prefix": true,
          "geohash_precision": 10
        }
      }
    }
  }
}

```

地理散列单元可以通过所有格式的地理点定义。如果单元通过经纬度值定义，需要设

置单元的大小。通过过滤中的 `precision` 参数进行定义, 设置为一个整数值表示地理散列前缀的长度。除了直接设置地理散列长度, 也可以用距离来定义精确度。例如 "precision": "50m"。

```
{
  "bool" : {
    "must" : { "match_all" : {} },
    "filter" : {
      "geohash_cell" : {
        "pin" : { "lat" : 13.4080, "lon" : 52.5186 },
        "precision" : 3,
        "neighbors" : true
      }
    }
  }
}
```

4.2.7 跨度查询

跨度查询是低级别的按位查询, 在指定索引词的顺序和接近度上提供专门的控制。这些查询通常用于在法律或专利文档上执行非常具体的查询。

跨度查询不能与非跨度查询混合(多索引词跨度查询例外)。

1. 索引词跨度查询

匹配包含索引词的跨度, 对应 Lucene 中的 `SpanTermQuery`:

```
{ "span_term" : { "user" : "kimchy" } }
```

增益也可以关联到查询中:

```
{ "span_term" : { "user" : { "value" : "kimchy", "boost" : 2.0 } } }
```

或者:

```
{ "span_term" : { "user" : { "term" : "kimchy", "boost" : 2.0 } } }
```

2. 多索引词跨度查询

可以封装多索引词查询(任何一个通配符、模糊、前缀、索引词、范围或正则表达式查询)为跨度查询, 所以可以被嵌套:

```
{
  "span_multi": {
    "match": { "prefix" : { "user" : { "value" : "ki", "boost" : 1.08 } } }
  }
}
```

3. 首跨度查询

匹配字段开始附近的跨度, 对应 Lucene 中的 `SpanFirstQuery`:

```
{
  "span_first" : {
    "match" : { "span_term" : { "user" : "kimchy" } },
    "end" : 3
  }
}
```

match 子句可以是任何其他类型的跨度查询。end 参数控制匹配中允许的最大结束位置。

4. 接近跨度匹配

匹配接近另一个的跨度。可以定义溢出值，表示介于不匹配位置之间的最大值。以及匹配是否按顺序提交请求，对应 Lucene 中的 SpanNearQuery:

```
{
  "span_near" : {
    "clauses" : [
      { "span_term" : { "field" : "value1" } },
      { "span_term" : { "field" : "value2" } },
      { "span_term" : { "field" : "value3" } }
    ],
    "slop" : 12,
    "in_order" : false,
    "collect_payloads" : false
  }
}
```

clauses 元素是一个或更多其他类型的跨度查询列表。

5. 或跨度查询

匹配它的跨度子句的并集，对应 Lucene 中的 SpanOrQuery:

```
{
  "span_or" : {
    "clauses" : [
      { "span_term" : { "field" : "value1" } },
      { "span_term" : { "field" : "value2" } },
      { "span_term" : { "field" : "value3" } }
    ]
  }
}
```

clauses 元素是一个或更多其他类型的跨度查询列表。

6. 非跨度查询

移除与另一个跨度查询重叠的匹配，对应 Lucene 中的 SpanNotQuery:

```
{
  "span_not" : {
    "include" : {
      "span_term" : { "field1" : "hoya" }
    },

```



```

"exclude" : {
  "span_near" : {
    "clauses" : [
      { "span_term" : { "field1" : "la" } },
      { "span_term" : { "field1" : "hoya" } }
    ],
    "slop" : 0,
    "in_order" : true
  }
}

```

include 和 exclude 子句可以是任何类型的跨度查询。include 子句的匹配会被过滤掉，exclude 子句返回的匹配必须没有重叠的部分。

7. 包含跨度查询

返回封装另一个跨度查询的匹配，对应 Lucene 中的 SpanContainingQuery:

```

{
  "span_containing" : {
    "little" : { "span_term" : { "field1" : "foo" } },
    "big" : {
      "span_near" : {
        "clauses" : [
          { "span_term" : { "field1" : "bar" } },
          { "span_term" : { "field1" : "baz" } }
        ],
        "slop" : 5,
        "in_order" : true
      }
    }
  }
}

```

big 和 little 子句可以是任何类型的跨度查询。从 big 匹配的跨度到包含从 little 返回的匹配。

8. 内部跨度查询

返回内部封装的另一个跨度查询的匹配，对应 Lucene 中的 SpanWithinQuery:

```

{
  "span_within" : {
    "little" : { "span_term" : { "field1" : "foo" } },
    "big" : {
      "span_near" : {
        "clauses" : [
          { "span_term" : { "field1" : "bar" } },
          { "span_term" : { "field1" : "baz" } }
        ],

```

```

        "slop" : 5,
        "in_order" : true
    }
}
}

```

big 和 little 子句可以是任何类型的跨度查询。从 little 匹配的跨度被封装在 big 返回的跨度中。

4.2.8 高亮显示

Elasticsearch 中的高亮显示是来源于 Lucene 的功能，允许在一个或者多个字段上突出显示搜索内容，Lucene 支持三种高亮显示方式 highlighter、fast-vector-highlighter、postings-highlighter，第一种是默认的标准类型。下面先看一个实例，在搜索前，先增加一条文档。

请求：PUT <http://127.0.0.1:9200/secilog/log/10?pretty>

参数：

```

{
  "type": "file",
  "message": "secilog is a log real-time analyse software, it's full text search
             is based on Elasticsearch "
}

```

文档创建好后，我们进行高亮搜索：

请求：POST http://127.0.0.1:9200/secilog/log/_search?pretty

参数：

```

{
  "query": {
    "term": { "message": "analyse" }
  },
  "highlight": {
    "fields": { "message": { } }
  }
}

```

返回值：

```

{
  "took" : 1,
  "timed_out" : false,
  "_shards" : { "total" : 1, "successful" : 1, "failed" : 0 },
  "hits" : {
    "total" : 1,
    "max_score" : 0.4232868,
    "hits" : [ {
      "_index" : "secilog",
      "_type" : "log",

```

```

    "_id" : "10",
    "_score" : 0.4232868,
    "_source" : {
      "type" : "file",
      "message" : "secilog is a log real-time analyse software,it's full
        text search is based on Elasticsearch "
    },
    "highlight" : {
      "message" : [ "secilog is a log real-time <em>analyse</em>
        software,it's full text search is based on Elasticsearch " ]//<em>表
        示高亮内容的关键字
      }
    }
  }
}

```

从结果中可以看出，有高亮显示的内容，analyse。为了执行高亮显示，该字段必须有实际的内容。并且这个字段必须进行存储，就是在字段映射中 store 的值必须为 true，不能只在内存中。系统会自动加载 _source 字段并匹配相关的列。字段名称支持通配符号，例如可以用 "message*": { } 参数匹配所有 message 开头的字段。

fast-vector-highlighter

highlighter 是普通的高亮显示，而 fast-vector-highlighter 高亮显示更加有特点，如下所示：

- ❑ 快，特别是内容大的字段，比如大于 1MB。
- ❑ 可定制的 boundary_chars, boundary_max_scan, 和 fragment_offset。
- ❑ 可以设置 term_vector 的值为 with_positions_offsets, 增加索引的大小。
- ❑ 可以将多个字段的匹配组合成一个结果。
- ❑ 可以权重匹配分配在不同的位置上。

在 Elasticsearch 中建立索引的时候映射字段类型，才可以实现 postings-highlighter 高亮显示，例如对 content 字段采用 fast-vector 高亮类型：

```

{
  "type_name" : {
    "content" : { "type": "string", "term_vector" : "with_positions_offsets" }
  }
}

```

postings-highlighter 高亮显示是另一种特殊功能，具有如下特点：

- ❑ 快，因为它不需要重新分析文档：尤其是对大文件对性能的提高更为明显。
- ❑ 占用更少的磁盘空间。
- ❑ 把高亮显示和句子分开，这个更有利于人类的阅读。
- ❑ 使用 BM25 算法，使搜索的时候像是整篇文档。

Elasticsearch 中需要在建立索引的时候映射字段类型，才可以实现 postings-highlighter 高亮显示，例如对 content 字段采用 postings 高亮类型：

```
{
  "type_name" : {
    "content" : {"type":"string","index_options" : "offsets"}
  }
}
```



提示 高亮查询不支持复杂的查询，比如查询类型设置为 `match_phrase_prefix` 的查询。

对于后两种特殊的高亮功能，都会增加索引的大小，但对于高亮显示查询的执行时间会减少。

使用 `type` 字段可以强制使用特定的高亮类型，当设置了 `term_vectors` 高亮类型的时候，想用普通高亮显示的时候也非常有用。这里只有三种类型，`plain`、`postings`、`fvh` 分别对应高亮显示的三种类型，例如：

```
{
  "query" : {...},
  "highlight" : {
    "fields" : {
      "content" : {"type" : "plain"}
    }
  }
}
```

默认高亮显示 HTML 标记。

默认情况下，高亮显示的文本在 `` 和 `` 中。这可以通过设置 `pre_tags` 和 `post_tags` 进行修改，例如：

```
{
  "query" : {...},
  "highlight" : {
    "pre_tags" : ["<b>"],
    "post_tags" : ["</b>"],
    "fields" : {
      "_all" : {}
    }
  }
}
```

可以有多个标签用数组进行表示，按照“重要性”进行排序，例如：

```
{
  "query" : {...},
  "highlight" : {
    "pre_tags" : ["<tag1>", "<tag2>"],
    "post_tags" : ["</tag1>", "</tag2>"],
    "fields" : {"_all" : {}}
  }
}
```

```

    }
  }
}

```

系统对于这种情况有默认多个 `pre_tags`, 需要设置 `tags_schema` 为 `styled`, 默认 `post_tags` 为 ``, 默认多个 `pre_tags` 标签为: `<em class="hlt1">`, `<em class="hlt2">`, `<em class="hlt3">`, `<em class="hlt4">`, `<em class="hlt5">`, `<em class="hlt6">`, `<em class="hlt7">`, `<em class="hlt8">`, `<em class="hlt9">`, `<em class="hlt10">`。

当我们需要设置默认多个标签时, 如下所示:

```

{
  "query" : {...},
  "highlight" : {
    "tags_schema" : "styled",
    "fields" : {"content" : {}}
  }
}

```

每个字段都可以设置高亮显示的字符片 `fragment_size` 段大小 (默认为 100), 以及返回的最大片段数 `number_of_fragments` (默认为 5), 如果 `number_of_fragments` 值设置为 0 则片段产生, 当 `order` 设置为 `score` 时候可以按照评分进行排序。例如:

```

{
  "query" : {...},
  "highlight" : {
    "order" : "score",
    "fields" : {"content" : {"fragment_size" : 150, "number_of_fragments" : 3}}
  }
}

```

4.3 简化查询

Elasticsearch 提供了一套精简 API 来查看系统的状态, 官方的文档叫 `cat API`。主要是由于 Elasticsearch 默认接口返回的都是 JSON 格式的, 这种格式不利于人类的阅读, 所以搞出来一套 `cat API` 来进行简化。

每个命令都是以 `/_cat` 开头, 可以接收 `v` 参数得到详细输出; 可以通过 `help` 参数得到每列的帮助信息。可以通过 `h=` 参数名返回部分内容, 多个参数可以用逗号分开, 甚至可以支持通配符。例如:

请求: `GET http://127.0.0.1:9200/_cat/master?v`

返回值:

```

id          host          ip          node
jevOQqVQT_a_pAGqKAOp7w 127.0.0.1 127.0.0.1 Artemis

```

查询帮助实例:

请求：GET http://127.0.0.1:9200/_cat/master?help

id		node id
host	h	host name
ip		ip address
node	n	node name

可以对系统的大部分内部结构进行简化查询，比如索引信息、节点信息，这些内部结构简称指标，简化查询的指标见表 4-5。

表 4-5 简化查询指标

指标	说明
aliases	别名，关于别名的一些信息，包括过滤和路由等信息，可以在最后加上具体别名的名称单独查询某个具体的别名
allocation	每个数据节点拥有多少分片快照和它们使用的磁盘空间大小
count	节点的文档总数，如果后面加上索引名称，可以得到具体索引的文档数量
fielddata	在集群中的每个数据节点上正在使用的堆内存的大小
health	集群的健康情况
indices	节点下的索引信息，可以得到索引的分片、文档、存储大小等信息
master	显示 master 节点的概要信息，主要是 id、ip 和节点名称
nodeattrs	显示自定义节点属性
nodes	显示节点相关的信息
pending tasks	集群任务接口，和集群接口类似，参见集群任务文档
plugins	提供集群中插件安装的信息
recovery	正在进行和以前完成的索引分片的回收率
repositories	显示在集群中注册的快照库
thread pool	显示每个集群节点的线程池统计
shards	索引分片的使用情况。分片有很多的指标，可以用 help 查看
segments	每个索引分片中低水平段的信息
snapshots	显示仓库的快照信息

下面分别介绍其中的常用简化查询指标。

1. indices

indices 有非常多的参数可以选择，具体用 help 查看。例如：

请求：GET http://127.0.0.1:9200/_cat/indices?pri&v&h=health,index,prirep,docs.count,mt

返回值：

health	index	docs.count	mt	pri	mt
yellow	customer	5	0		0
green	.scripts	2	0		0
green	secilog	2	0		0
yellow	secisland	4	0		0

2. nodes

nodes 有非常多的参数可以选择, 具体用 help 查看。例如:

请求: GET http://127.0.0.1:9200/_cat/nodes?v&h=id,ip,port,v,m,jdk

返回值:

```
id ip port v m jdk
qKTT 127.0.0.1 9300 2.2.0 * 1.7.0_79
```

3. recovery

在一个集群中, 一个索引的分片可能随时会从一个节点迁移到另一个节点, 比如在恢复快照的过程中, 新节点启动的过程中, 等等。例如下面的示例表示没有任何分片的数据在传输。

请求: GET http://127.0.0.1:9200/_cat/recovery/secisland?v

返回值:

index	shard	time	type	stage	source_host	target_host	repository	snapshot	files
secisland	0	26	store	done	127.0.0.1	127.0.0.1	n/a	n/a	0
secisland	1	46	store	done	127.0.0.1	127.0.0.1	n/a	n/a	0
secisland	2	31	store	done	127.0.0.1	127.0.0.1	n/a	n/a	0
secisland	3	32	store	done	127.0.0.1	127.0.0.1	n/a	n/a	0
secisland	4	25	store	done	127.0.0.1	127.0.0.1	n/a	n/a	0

files_percent	bytes	bytes_percent	total_files	total_bytes	translog	translog_percent	total_translog
0.0%	0	0.0%	0	0	0	100.0%	0
0.0%	0	0.0%	0	0	0	100.0%	0
0.0%	0	0.0%	0	0	0	100.0%	0
0.0%	0	0.0%	0	0	0	100.0%	0
0.0%	0	0.0%	0	0	0	100.0%	0

通过增加副本节点的数量可以让分片的数据进行传输。

例如:

index	shard	time	type	stage	source	target	files	percent	bytes	percent
wiki	0	1252	store	done	hostA	hostA	4	100.0%	23638870	100.0%
wiki	0	1672	replica	index	hostA	hostB	4	75.0%	23638870	48.8%
wiki	1	1698	replica	index	hostA	hostB	4	75.0%	23348540	49.4%
wiki	1	4812	store	done	hostA	hostA	33	100.0%	24501912	100.0%
wiki	2	1689	replica	index	hostA	hostB	4	75.0%	28681851	40.2%
wiki	2	5317	store	done	hostA	hostA	36	100.0%	30267222	100.0%

4. thread pool

thread pool (线程池) 有非常多的参数可以选择, 具体用 help 查看。例如:

请求: GET http://127.0.0.1:9200/_cat/thread_pool?v&h=id,host,suggest.active,suggest.rejected,suggest.completed

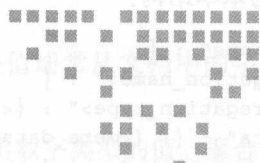
返回值:

```
id      host      suggest.active suggest.rejected suggest.completed
qKTT 127.0.0.1      0          0          0
```

4.4 小结

本章介绍 Elasticsearch 中搜索相关的知识,包括通过参数查询和 DSL 语言进行查询。通过本章可以了解到查询的内部知识和相关的高级用法。同时 Elasticsearch 还提供了简化查询方法来简化查询。

5.2 度量聚合



第5章 Chapter 5

聚 合

聚合是一种基于搜索的数据汇总，通过组合可以完成复杂的操作。聚合可以对文档中的数据进行统计汇总、分组等，对一个聚合进行操作可以看作在一组文档中分析数据。本章主要介绍聚合相关的知识，包括聚合的分类、度量聚合、分组聚合和管道聚合。

5.1 聚合的分类

系统中有不同类型的聚合，每种都有特定的用途和输出结果。为了更好地理解这些类型，通常将聚合分为三大类：

- **度量聚合**——在一组文档中对某一个数字型字段进行计算得出指标值。
- **分组聚合**——创建多个分组，每个分组都关联一个关键字和相关文档标准。当聚合执行的时候，所有的分组会根据自身标准评估每一个符合的文档。当文档匹配分组标准的时候，会将文档划分进相关的分组里。聚合进行完成时，会给出一个分组列表，每一个分组都会有一组“属于”它的文档。
- **管道聚合**——这一类聚合的数据源是其他聚合的输出，然后进行相关指标的计算。

除此之外，还可以在分组聚合的基础上添加聚合并执行，这就是聚合的嵌套，通过嵌套可以完成很多复杂的聚合操作！



提示 分组聚合可以拥有子聚合（分组聚合或者度量聚合），子聚合将计算由父聚合生成的分组。对于聚合嵌套的水平和深度没有硬性的限制（可以在一个“父”聚合下嵌套一个或多个子聚合）。

聚合操作的基本结构：

```
"aggregations" : {
  "<aggregation_name>" : {
    "<aggregation_type>" : {<aggregation_body>}
    [,"meta" : { [ <meta_data_body> ] } ]?
    [,"aggregations" : { [ <sub_aggregation> ]+ } ]?
  }
  [,"<aggregation_name_2>" : { ... } ]*
```

聚合操作用 `aggregations` (也可以简化为 `aggs`) 关键字作为开头, 操作的对象是 JSON 数据结构。每一个聚合关联一个用户定义的逻辑名 (即上面代码中的 `<aggregation_name>` 标签) (例如, 如果聚合计算平均价格, 可能命名为 `avg_price`)。这些逻辑名也用来唯一地标识响应里面的聚合结果。每一个聚合拥有一个特定类型 (即上面代码中的 `<aggregation_type>` 标签), 通常作为聚合的第一个关键字。每一种类型的聚合会根据聚合的性质定义参数 (例如, 在一个特定字段上添加平均值聚合, 将在这个字段上面进行平均值的计算)。当定义了同一级别的聚合且类型是分组聚合的时候, 可以定义一组额外的聚合。在这种情况下, 在分组聚合的级别上定义的子聚合会计算所有的分组聚合执行产生的分组。举个例子, 如果在范围聚合下定义了一组聚合, 子聚合会计算所有的范围聚合产生的分组。

聚合值的数据源: 一些聚合针对参与聚合的文档中提取出来的值进行计算。通常情况下, 这些值是从特定的文档字段中提取出来的。也可以定义一个脚本, 针对每一个文档生成这些值。

当一个聚合里面同时包含文档字段和脚本时, 脚本就会被视为值脚本。普通脚本在文档级别上计算值 (即脚本需要访问所有与文档相关联的数据), 值脚本在值的级别上进行计算。在这种模式下, 从配置的字段中提取值并使用脚本对这些值进行“转换”。



提示 当使用脚本的时候, 依然可以定义 `lang` 和 `params` 设置。前者定义了脚本使用的语言 (假设适当的语言在 `Elasticsearch` 中可见, 无论是作为默认语言还是作为插件)。后者可以在脚本中把所有的“动态”表达式定义为参数, 这样就可以确保脚本在调用期间保持不变 (这将确保缓存编译的脚本在 `Elasticsearch` 中的可用性)。

脚本可以针对每个文档生成一个或多个值。当生成多个值的时候, 可以使用 `script_values_sorted` 设置来表明这些值是否需要排序。本质上, `Elasticsearch` 处理排序过的值时可以执行优化 (例如, 执行最小值聚合的时候, 如果知道这些值是排序过的, `Elasticsearch` 就会跳过所有值的计算过程, 确定列表中的第一个值就是所有文档中最小值)。

5.2 度量聚合

度量聚合从文档中提取出来值并进行计算。这些值通常从文档中的字段（使用数据字段）中提取出来，但也可以使用脚本进行计算。

数字型度量聚合是一种特殊类型的度量聚合，输出数字类型的值。聚合输出一个数字指标（例如平均值聚合）称为**单值数字型度量聚合**，产生多个指标值（例如统计聚合）称为**多值数字型度量聚合**。当这些聚合直接作为一些分组聚合的子聚合时，单值和多值数字型度量聚合的内容就会发挥巨大的作用，例如分组聚合可以对度量聚合后的返回的值进行排序。

5.2.1 平均值聚合

平均值聚合是一个单值度量聚合，计算从聚合的文档中提取的数字型值的平均值。这些值既可以从文档中的数字型字段中提取，也可以通过脚本生成。

假设数据由代表学生考试成绩（从 0 到 100）的文档组成。

```
{
  "aggs" : {    // 聚合的名称
    "avg_grade" : { "avg" : { "field" : "grade" } }
  }
}
```

上面的聚合通过对所有文档计算平均成绩。聚合类型是 avg，field 设置定义了文档中需要计算平均数的数字型字段。

```
{
  ...
  "aggregations": {
    "avg_grade": {    // 返回结果中包括聚合名称
      "value": 75
    }
  }
}
```

执行聚合，将返回类似上面的数据。聚合的名称为 avg_grade，这个名称在请求中进行了定义，在返回响应的结果中作为关键字返回，里面的内容为聚合后的结果。

1. 脚本

基于脚本计算平均成绩：

```
{
  ...,
  "aggs" : { "avg_grade" : { "avg" : { "script" : "doc['grade'].value" } } }
}
```

上面的内容将会用系统内置的脚本语言进行解释，而且是不带参数的。使用一个文件脚本的完整语法如下：

```


{
  "aggs" : {
    "avg_grade" : {
      "avg" : {
        "script" : {
          "file": "my_script",
          "params": {"field": "grade"}
        }
      }
    }
  }
}

```

聚合操作的基本结构:

聚合操作的基本结构:

聚合操作的基本结构:

 **提示** 要想执行被索引脚本只需要将 file 参数替换为 id 参数。

2. 文档值脚本

可以对文档中具体某列的值用脚本进行再次计算:

```

{
  "aggs" : {
    ...
    "avg" : {
      "avg" : {
        "field": "grade",
        "script" : {
          "inline": "_value * correction", // 脚本的语句
          "params": {
            "correction": 1.2 // 参数值
          }
        }
      }
    }
  }
}

```

聚合操作的基本结构:

聚合操作的基本结构:

聚合操作的基本结构:

3. 默认值

Missing 字段定义了文档缺失值时应该如何处理。默认情况下这些文档会被忽略,但是也可以设置一个默认值:

```

{
  "aggs" : {
    "grade_avg" : {
      "avg" : {"field": "grade", "missing": 10}
    }
  }
}

```

聚合操作的基本结构:

聚合操作的基本结构:

聚合操作的基本结构:

```
}
}
```

当 grade 字段没有值的时候，系统会用一个默认值 10 来作为计算的值。

5.2.2 基数聚合


这是一个单值度量聚合，计算不同值的近似计数。值可以从文档的特定字段中提取，也可以通过脚本生成。

假设你在索引书籍而且想要统计符合条件的作者，代码如下：

```
{
  "aggs" : {
    "author_count" : {
      "cardinality" : { "field" : "author" }
    }
  }
}
```

聚合支持 precision_threshold 选项：

```
{
  "aggs" : {
    "author_count" : {
      "cardinality" : {
        "field" : "author_hash",
        "precision_threshold": 100
      }
    }
  }
}
```

 **警告** precision_threshold 选项是基数聚合当前特别的内部实现，这可能会在将来的版本中改变。

precision_threshold 选项允许为了准确性而设置交换内存，并且定义了一种独特的计数。低于计数将接近准确，高于这个值计数可能会变得更模糊。最大值为 40000，阈值高于这个数，产生的效果和阈值为 40000 时相同。默认值取决于创造多分组的父聚合数量。

1. 计数是近似值

计算精确计数要求加载值到一个哈希集并且返回它的大小，当工作在高基数集时需要占用大量的内存空间，同时在节点间沟通每个分片集也会占用大量的集群资源，所以计算的效率是比较低的。

基数聚合基于 HyperLogLog++ 算法。

2. 脚本

基数聚合指标支持脚本，不过使用脚本会带来明显的性能损失。

```
{
  "aggs" : {
    "author_count" : {
      "cardinality" : {
        "script": "doc['author.first_name'].value + ' ' + doc['author.last_name'].value"
      }
    }
  }
}
```

上面的内容将会用系统内置的脚本语言进行解释，而且是不带参数的。用一个文件脚本的完整语法如下：

```
{
  "aggs" : {
    "author_count" : {
      "cardinality" : {
        "script" : {
          "file": "my_script",
          "params": { // 具体参数的值
            "first_name_field": "author.first_name",
            "last_name_field": "author.last_name"
          }
        }
      }
    }
  }
}
```



提示 索引脚本只需要将 file 参数替换为 id 参数。

3. 默认值

Missing 字段定义了文档缺失值的时候应该如何处理。默认情况下这些文档会被忽略，但是也可以认为它们有一个默认的值存在：

```
{
  "aggs" : {
    "tag_cardinality" : { // 关键字
      "cardinality" : {
        "field" : "tag",
        "missing": "N/A" // 默认值
      }
    }
  }
}
```

```
}
}
```

表示 tag 字段中没有内容的文档会划入相同的分组里, 这些分组具有相同的默认值 N/A。

5.2.3 最大值聚合

最大值聚合是一个单值度量聚合, 记录和返回从聚合的文档中提取出的数字型值中的最大值。

这些值可以从文档的特定数字型字段提取, 也可以通过脚本生成。

通过所有文档计算最高价格的语句如下:

```
{
  "aggs" : { "max_price" : { "max" : { "field" : "price" } } }
}
```

返回值:

```
...
"aggregations": {
  "max_price": {
    "value": 35
  }
}
```

可以看出, 聚合的名称 (max_price 请求时定义) 作为关键字可以通过它从返回的响应中搜索出聚合的结果。


1. 脚本

通过脚本计算最高价格:

```
{
  "aggs" : { "max_price" : { "max" : { "script" : "doc['price'].value" } } }
}
```

这会将脚本参数解释为一个用默认脚本语言写的却没有脚本参数的内联脚本:

```
{
  "aggs" : {
    "max_price" : {
      "max" : {
        "script" : {
          "file": "my_script",
          "params": { "field": "price" }
        }
      }
    }
  }
}
```


 **提示** 索引脚本只需要将 file 参数替换为 id 参数。

2. 值脚本

假设我们需要换算币种之间的汇率，我们可以利用值脚本对每个值在聚合之前应用汇率：

```
{
  "aggs" : {
    "max_price_in_euros" : {
      "max" : {
        "field" : "price",
        "script" : {
          "inline": "_value * conversion_rate",
          "params" : {"conversion_rate" : 1.2}
        }
      }
    }
  }
}
```

5.2.4 最小值聚合

最小值聚合是一个单值度量聚合，记录和返回从聚合的文档中提取出的数字型值中的最小值。这些值可以从文档的特定数字型字段提取，也可以通过脚本生成。

通过所有文档计算最低价格：

```
{
  "aggs" : { "min_price" : { "min" : { "field" : "price" } } }
}
```

返回值：

```
{
  ...
  "aggregations": {
    "min_price": { "value": 10 }           // 聚合的名称
  }
}
```

可以看出，聚合的名称（min_price 请求时定义）作为关键字在响应的结果中返回。

5.2.5 和聚合

和聚合是一个单值度量聚合，对聚合文档中提取的数字型值进行求和。这些值可以从文档的特定数字型字段提取，也可以通过脚本生成。脚本的语法和前面介绍的类似，就不具体介绍了，语法如下：

```
"aggs" : {"intraday_return" : { "sum" : { "field" : "change" } }}}
```

5.2.6 值计数聚合

值计数聚合是一个单值度量聚合，对聚合文档中提取的值进行计数。这些值可以从文档的特定数字型字段提取，也可以通过脚本生成：

```
{
  "aggs" : {"grades_count" : { "value_count" : { "field" : "grade" } }}
}
```

通常情况下，这个聚合会与其他单值聚合一块使用。例如，当计算平均值聚合时，我们可能希望知道有多少值参与了平均运算。

5.2.7 统计聚合

统计聚合是一个多值度量聚合，对聚合文档中提取的数字型值进行统计计算。这些值可以从文档的特定数字型字段提取，也可以通过脚本生成。脚本的语法和前面介绍的类似，就不具体介绍了，语法如下：

```
{
  "aggs" : {"grades_stats" : { "stats" : { "field" : "grade" } }}
}
```

统计聚合包含：最小值、最大值、和、计数、平均数聚合。

5.2.8 百分比聚合

百分比聚合是一个多值度量聚合，对聚合文档中提取的数字型值计算一个或多个百分比。这些值可以从文档的特定数字型字段提取，也可以通过脚本生成。

百分比聚合得到的点是在特定比例下出现的观测值。例如，95 百分比对应的值表示这个值大于 95% 的所有值。

百分比聚合经常用于寻找极端值。在正态分布中，0.13% 和 99.87% 代表三个标准偏差的平均值。任何落在三个标准偏差之外的数据通常被认为是出现了异常。

我们可以利用百分比聚合的结果评估数据分布，判断数据是否扭曲，分析数据是否符合双峰分布等。

假设你的数据是由网站加载时间组成的。对于管理员来说，相对于加载时间的平均值和中间值，更加感兴趣的是最大值。但是一条慢响应可轻易影响这个值。

我们可以查询代表加载时间的百分比聚合：

```
{
  "aggs" : {
    "load_time_outlier" : {
      "percentiles" : { "field" : "load_time" }
    }
  }
}
```

```

    }
  }
}

```

其中, `load_time` 字段必须是数字型字段。

默认情况下, 百分比指标会计算一系列百分比: [1, 5, 25, 50, 75, 95, 99]。响应看起来会是这样:

```

{
  ...
  "aggregations": {
    "load_time_outlier": {
      "values": {
        "1.0": 15,
        "5.0": 20,
        "25.0": 23,
        "50.0": 25,
        "75.0": 29,
        "95.0": 60,
        "99.0": 150
      }
    }
  }
}

```

正如我们看到的, 聚合会对每个百分比返回一个计算过的值。假设响应时间是毫秒级, 立即就能得出通常情况下 75% 的网页加载时间是 15 ~ 29ms, 但是偶尔会达到 60 ~ 150ms, 99% 的网页在 150ms 内加载完成。

通常, 管理员只对极端值感兴趣——极端的百分比。我们可以仅仅指出我们感兴趣的百分比 (请求的百分比必须是在 0 ~ 100 之间的值)。

```

{
  "aggs": {
    "load_time_outlier": {
      "percentiles": { "field": "load_time", "percents": [95, 99, 99.9] }
    }
  }
}

```

利用 `percents` 参数指定的百分比来进行计算。

5.2.9 百分比分级聚合

百分比分级聚合是一个多值度量聚合, 对聚合文档中提取的数字型值计算一个或多个级别的百分比。这些值可以从文档的特定数字型字段提取, 也可以通过脚本生成。

百分比等级表示测试值低于某一特定值的百分比。例如, 如果一个值大于或者等于所有测试值的 95%, 就表示这个值在第 95 百分比等级。

假设你的数据由网站加载时间组成。你有一个服务协议，规定 95% 的页面在 15ms 内完成加载，99% 的页面在 30ms 内完成加载。

我们可以查询加载时间的百分比等级，如下所示：

```
{
  "aggs" : {
    "load_time_outlier" : {
      "percentile_ranks" : {
        "field" : "load_time",
        "values" : [15, 30]
      }
    }
  }
}
```

其中，Load_time 字段必须是数字型字段。

响应看起来是这样的：

```
{
  ...
  "aggregations": {
    "load_time_outlier": {
      "values" : {"15": 92, "30": 100}
    }
  }
}
```

从这些信息可以看出来，100% 的页面在 30ms 内加载，92% 的页面在 15ms 内加载，即实现了 99% 的页面在 30ms 内加载的目标，但没有完成 95% 页面在 15ms 内加载的目标。

5.2.10 最高命中排行聚合

最高命中排行聚合会在聚合文档中找出相关度最高的文档。这个聚合用来作为子聚合使用，这样每个分组中相关度最高的文档就可以聚合在一起。

选项参如下所示：

- ❑ from——第一个结果的偏移量。
- ❑ size——每个分组返回的命中文档的最大数量。默认返回前三个命中文档。
- ❑ sort——指定最高命中文档应该如何排序。默认情况下按照主查询的分数排序。

5.2.11 脚本度量聚合



警告 这个功能是实验性的，可能在未来的版本中完全改变或移除。

利用脚本执行的度量聚合，提供一个指标输出。

例 5-1 脚本度量聚合

```

{
  "query" : {
    "match_all" : {}
  },
  "aggs" : {
    "profit" : {
      "scripted_metric" : {
        "init_script" : "_agg['transactions'] = []",
        "map_script" : "if (doc['type'].value == \"sale\") { _agg.transactions.add(doc['amount'].value) } else { _agg.transactions.add(-1 * doc['amount'].value) }",
        "combine_script" : "profit = 0; for (t in _agg.transactions) { profit += t }; return profit",
        "reduce_script" : "profit = 0; for (a in _aggs) { profit += a }; return profit"
      }
    }
  }
}

```

map_script 是唯一必要的参数。上面的聚合演示了如何使用脚本聚合通过销售和交易成本来计算总利润。

可能的响应是：

```

{
  ...
  "aggregations" : {
    "profit" : { "value" : 170 }
  }
}

```

也可以指定使用的文件脚本：

```

{
  "query" : { "match_all" : {} },
  "aggs" : {
    "profit" : {
      "scripted_metric" : {
        "init_script" : { "file" : "my_init_script" },
        "map_script" : { "file" : "my_map_script" },
        "combine_script" : { "file" : "my_combine_script" },
        "params" : { "field" : "amount" },
        "reduce_script" : { "file" : "my_reduce_script" },
      }
    }
  }
}

```

其中，Init、map 和 combine 脚本的参数必须在全局 params 对象中指定，这样才可以在

脚本间共享参数。

1. 允许的返回值类型

当一个脚本同时包含多个有效脚本对象时，脚本对象必须返回或存储以下类型的 `_agg` 对象：

- ☐ 原始类型
- ☐ 字符串
- ☐ 图（只包含这里列出的类型的键和值）
- ☐ 数组（只包含这里列出的类型的元素）

2. 脚本的作用域

脚本度量聚合的执行有 4 个阶段。

1) 初始化脚本 (`init_script`)。在任何文档的收集之前执行。允许聚合设置任何初始化状态。

在例 5-1 里，初始化脚本在 `_agg` 对象中创建了一个名为 `transactions` 的数组。

2) 映射脚本 (`map_script`)。每个被采集的文档都会执行一次脚本。这是唯一必须的脚本。如果没有指定联合脚本，结果状态需要存储在一个名为 `_agg` 的对象中。

在例 5-1 中，映射脚本检查类型字段的值。如果值为销售 (`sale`)，就把数量字段的值加到事务数组里面。如果值不是销售，就把数量字段的值取负数添加到事务数组里。

3) 联合脚本 (`combine_script`)。每个分片会在文档采集结束的时候执行一次脚本。允许聚合从每个分片中统一状态。如果没有提供联合脚本，联合阶段就会返回聚合变量。

在例 5-1 中，联合脚本迭代所有存储的事务数组，求出利润变量的值，最后返回利润。

4) 归纳脚本 (`reduce_script`)。在所有分片返回结果之后，请求节点执行一次脚本。这个脚本用于访问 `_aggs` 变量，`_aggs` 变量是每个分片执行联合脚本之后的结果数组。如果没有提供归纳脚本，归纳阶段会返回 `_aggs` 变量。

在例 5-1 中，归纳脚本迭代每个分片返回的利润，求出最终在聚合响应中返回的合并利润。

例 5-2

试想这样的情形，你在两个分片中索引下面的文档：

```
PUT http://127.0.0.1:9200/transactions/stock/1
{"type": "sale", "amount": 80}
PUT http://127.0.0.1:9200/transactions/stock/2
{"type": "cost", "amount": 10}
PUT http://127.0.0.1:9200/transactions/stock/3
{"type": "cost", "amount": 30}
PUT http://127.0.0.1:9200/transactions/stock/4
{"type": "sale", "amount": 130}
```

我们规定文档 1 和 3 在分片 A 上面, 文档 2 和 4 在分片 B 上面。然后, 我们可以分析上面的例子里每一步聚合的结果。

初始化脚本执行之前, 没有指定参数对象, 所以使用默认的参数对象。

```
"params" : { "_agg" : {} }
```

初始化脚本执行之后, 在每个分片开始执行文档采集之前, 运行初始化脚本, 我们会在每个分片上得到一个数组的拷贝。

分片 A:

```
"params" : {
  "_agg" : { "transactions" : [] }
}
```

分片 B:

```
"params" : {
  "_agg" : { "transactions" : [] }
}
```

映射脚本执行之后, 每个分片采集自身的文档, 并在采集到的文档上执行映射脚本。

分片 A:

```
"params" : {
  "_agg" : { "transactions" : [ 80, -30 ] }
}
```

分片 B:

```
"params" : {
  "_agg" : { "transactions" : [ -10, 130 ] }
}
```

联合脚本执行之后:

❑ 每个分片在文档采集完成之后会执行联合脚本, 将交易额换算成一个利润数字 (通过事物数组里的值相加得到), 返回给请求节点。

❑ 分片 A: 50。

❑ 分片 B: 120。

归纳脚本执行之后:

❑ 归纳脚本收到一个 `_aggs` 数组, 包含各个节点执行联合脚本的结果:

```
"_aggs" : [ 50, 120 ]
```

❑ 归纳脚本将所有分片的响应换算为最终的整体利润额 (通过值相加获得), 然后将其作为聚合的结果构造响应并返回:

```
{
  ...
  "aggregations": {
```

```

    "profit": {"value": 170}
  }
}

```

其他参数如下：

□ **params**——可选参数。可以作为参数传递给初始化脚本、映射脚本和联合脚本的对象，用于控制聚合行为和存储脚本执行的中间状态。如果没有指定，相当于提供了一个默认值：

```
"params" : {"_agg" : {}}
```

□ **reduce_params**——可选参数。可以作为参数传递给归纳脚本的对象，用于控制归纳阶段的行为。如果没有指定，这个变量在归纳脚本执行的时候会未定义。

5.2.12 地理边界聚合

地理边界聚合是一个度量聚合，为一个字段计算包含所有地点值的边界框。

示例：

```

{
  "query" : {"match" : { "business_type" : "shop" }},
  "aggs" : {
    "viewport" : {
      "geo_bounds" : { "field" : "location", "wrap_longitude" : true }
    }
  }
}

```

地理边界聚合指定一个字段用来获得边界。

wrap_longitude 是一个可选参数，指定边界框是否允许与国际日期变更线重叠。默认值是 **true**。

上面的聚合展示了查询所有业务类型为商店的文档，并且根据文档中的位置字段计算出边界框。

返回值：

```

{
  ...
  "aggregations": {
    "viewport": {
      "bounds": {
        "top_left": { // 左上角坐标
          "lat": 80.45,
          "lon": -160.22
        },
        "bottom_right": { // 右下角坐标
          "lat": 40.65,
          "lon": 42.57
        }
      }
    }
  }
}

```

5.2.13 地理重心聚合

地理重心聚合是一个度量聚合，从文档中的地理点数据类型字段获取的所有坐标值中计算出有利的矩心。

示例：

```
{
  "query" : { "match" : { "crime" : "burglary" } },
  "aggs" : {
    "centroid" : {
      "geo_centroid" : { "field" : "location" }
    }
  }
}
```

地理重心聚合指定字段用来计算矩心（注意，字段必须是地理点数据类型）。

上面的聚合展示了查询所有犯罪类型为盗窃的文档，并根据文档中的位置字段计算出矩心。

返回值：

```
{
  ...
  "aggregations": {
    "centroid": {
      "location": { // 地理矩心位置
        "lat": 80.45,
        "lon": -160.22
      }
    }
  }
}
```

地理重心聚合常作为子聚合与其他分组聚合结合起来使用。

示例：

```
{
  "query" : { "match" : { "crime" : "burglary" } },
  "aggs" : {
    "towns" : {
      "terms" : { "field" : "town" },
      "aggs" : {
        "centroid" : { "geo_centroid" : { "field" : "location" } }
      }
    }
  }
}
```

```

    }
  }
}

```

上面的示例用地理重心聚合作为索引分组聚合的子聚合来找出每个城镇中犯盗窃罪最严重的位置。

返回值:

```

{
  ...
  "buckets": [
    {
      "key": "Los Altos", // 城镇 Los Altos 犯罪最严重的位置
      "doc_count": 113,
      "centroid": {
        "location": {
          "lat": 37.3924582824111,
          "lon": -122.12104808539152
        }
      }
    },
    {
      "key": "Mountain View", // 城镇 Mountain View 犯罪最严重的位置
      "doc_count": 92,
      "centroid": {
        "location": {
          "lat": 37.382152481004596,
          "lon": -122.08116559311748
        }
      }
    }
  ]
}

```

5.3 分组聚合

分组聚合不像度量聚合那样通过字段进行计算，而是根据文档创建分组。每个聚合都关联一个标准（取决于聚合的类型），决定了一个文档在当前的条件下是否会“划入”分组中。换句话说，分组实际上定义了一个文档集。除了这些分组之外，分组聚合也会计算和返回“划入”每个分组中文档的数量。

与度量聚合不同，分组聚合可以拥有子聚合。这些子聚合可以聚合由它们的“父”聚合创建的分组。

分组聚合有不同的类型，对应着不同的“分组”策略。有的策略定义一个分组（单分组聚合），有的策略定义固定数量的多个分组（多分组聚合），还有的策略在聚合执行过程中动

态地创建分组。

5.3.1 子聚合

子聚合是一个特殊的单分组聚合，可以通过父类型文档的分组聚合产生子类型文档的分组。这种聚合依赖于映射中的 `_parent` 字段，只有一个选项：`type` 表示父空间的分组应该被映射为哪一种子类型。

例如，我们有一个包含问题和答案两个类型文档的索引。答案类型在映射中有 `_parent` 字段：

```
{
  "answer" : {
    "_parent" : {"type" : "question"}
  }
}
```

问题类型的文档拥有一个标签字段，答案类型的文档拥有一个所有者字段。即使两个字段分别存在于两种不同类型的文档中，通过子聚合也可以把一个问题类型文档的标签分组映射到答案类型文档的所有者分组。

例如一个问题类型的文档：

```
{
  "body": "<p>I have Windows 2003 server and i bought a new Windows 2008 server...",
  "title": "Whats the best way to file transfer my site from server to a newer one?",
  "tags": [
    "windows-server-2003",
    "windows-server-2008",
    "file-transfer"
  ],
}
```

一个答案类型的文档：

```
{
  "owner": {
    "location": "Norfolk, United Kingdom",
    "display_name": "Sam",
    "id": 48
  },
  "body": "<p>Unfortunately your pretty much limited to FTP...",
  "creation_date": "2009-05-04T13:45:37.030"
}
```

利用子聚合把两者结合在一起：

```
{
  "aggs": {
    "top-tags": {
      "terms": { // 父问题的 tags 分组
```

```

    "field": "tags",
    "size": 10
  },
  "aggs": {
    "to-answers": {
      "children": { // 子答案分组
        "type": "answer"
      },
      "aggs": {
        "top-names": {
          "terms": { // 子答案的 owner.display_name 字段分组
            "field": "owner.display_name",
            "size": 10
          }
        }
      }
    }
  }
}

```

type 指向名为 answer 的类型 / 映射。

这个示例返回置顶的问题标签以及每个标签下置顶答案的所有者。

可能得到的返回值：

```

{
  "aggregations": {
    "top-tags": {
      "buckets": [
        {
          "key": "windows-server-2003", // windows-server-2003 的问题
          "doc_count": 25365,
          "to-answers": {
            "doc_count": 36004,
            "top-names": {
              "buckets": [ // 答案中的 owner.display_name 字段分组结果
                { "key": "Sam", "doc_count": 274 },
                { "key": "chris", "doc_count": 19 },
                ...
              ]
            }
          }
        },
        {
          "key": "linux",
          "doc_count": 18342,
          "to-answers": {
            "doc_count": 6655,
            "top-names": {
              "buckets": [

```

```

    {"key": "abrams", "doc_count": 25},
    {"key": "ignacio", "doc_count": 25},
  ]
}

```

5.3.1 子聚合

子聚合是一个特殊的单分组聚合，可以通过聚合函数中的 `sub_aggs` 来指定。子聚合依赖于映射中的 `parent` 字段，且只有一个选项 `type` 来指定子聚合应该被映射为哪一种子类型。

5.3.2 直方图聚合

直方图聚合是一个多分组聚合，可以应用于从文档中提取的数值。在数值上动态创建固定大小（又名区间）的分组。例如，如果文档中有一个字段存储价格（数字型），我们可以配置聚合来动态生成间隔为 5 的分组（代表价格的情况下，意思是 5 元）。当聚合执行的时候，每个文档的价格字段会进行评估并划入与之最接近的分组里。举例说明，如果价格是 32 而且分组的大小是 5，那么取整之后，这些文档会“划入”与 30 相关联的分组。

取整算法如下：

```

rem = value % interval
if (rem < 0) {
    rem += interval
}
bucket_key = value - rem

```

从取整算法中我们可以看出，间隔本身必须是整数。



警告 当前版本中，在分组之前值会被转换为整数，这会造成负的浮点数值划入错误的分组中。例如，-4.5 和间隔为 2 的分组，-4.5 会被转换为 -4，并且最终会划入 $-4 \leq \text{val} < -2$ 的分组而不是 $-6 \leq \text{val} < -4$ 的分组。

举例说明基于价格的产品分配到区间为 50 的分组中：

```

{
  "aggs" : {
    "prices" : {
      "histogram" : {"field" : "price", "interval" : 50}
    }
  }
}

```

返回值：

```

{
  "aggregations": {
    "prices": {
      "buckets": [

```

```

    { //0 到 50 的数量
      "key": 0,
      "doc_count": 2
    },
    { //50 到 100 的数量
      "key": 50,
      "doc_count": 4
    },
    { //100 到 150 的数量
      "key": 100,
      "doc_count": 0
    },
    { //150 到 200 的数量
      "key": 150,
      "doc_count": 3
    }
  ]
}

```

1. 最小文档计数

上面的响应展示了没有价格区间为 [100 - 150] 的文档划入分组中。默认情况下，响应会用空分组填补直方图中的空白。可以利用 `min_doc_count` 设置来修改分组，要求一个更高的最低计数：

```

{
  "aggs" : {
    "prices" : {
      "histogram" : {
        "field" : "price",
        "interval" : 50,
        "min_doc_count" : 1 // 最少要有一个文档
      }
    }
  }
}

```

返回值：

```

{
  "aggregations": {
    "prices": {
      "buckets": [
        {"key": 0, "doc_count": 2},
        {"key": 50, "doc_count": 4},
        { //50 到 100 之间没有记录则不显示
          "key": 150,
          "doc_count": 3
        }
      ]
    }
  }
}

```

```

    ]
  }
}
}

```

默认情况下，直方图返回的分组范围基于数据本身，就是说，拥有最小值的文档（参与直方图的构成）决定了最小的分组（拥有最小的键值）；拥有最大值的文档决定了最大的分组（拥有最大的键值）。通常，请求空分组的时候，会导致混乱，特别是在数据被过滤的情况下。

为了理解这些，我们来分析一个例子。

例 5-3 假设我们请求过滤所有文档，要求值介于 0 到 500 之间，此外我们还要用间隔为 50 的直方图划分价格数据。同时，为了获取所有的分组（即使是空分组），指定“min_doc_count”参数为 0。

如果发生了这种情况——所有文档的价格均大于 100，我们获得的第一个分组就会以 100 作为它的键值。这就很令人困扰，在许多时候，我们也需要获得那些在 0 ~ 100 之间的分组。

利用 extended_bounds 设置，我们现在可以“强制”直方图聚合从指定的最小值开始创建分组，直到最大值（即使没有任何文档存在）。只有在 min_doc_count 参数为 0 的时候，extend_bounds 参数才有意义，当 min_doc_count 参数大于 0 时，永远不会返回空分组。

注意，extended_bounds 不会过滤分组。意味着，如果 extended_bounds.min 比文档中提取的值要大的时候，文档依然决定创建的第一个分组是什么（对于最后一个分组和 extended_bounds.max 来说，也是一样的情况）。为了过滤分组，我们应该利用适当的 from/to 设置把直方图聚合结合在一个范围过滤聚合中。

示例：

```

{
  "query" : { "constant_score" : { "filter" : { "range" : { "price" :
    { "to" : "500" } } } } } },
  "aggs" : {
    "prices" : {
      "histogram" : {
        "field" : "price",
        "interval" : 50,
        "extended_bounds" : { "min" : 0, "max" : 500 }
      }
    }
  }
}

```

2. 排序

返回的分组默认按照它们的键升序排序，我们可以用 order 设置来控制排序行为。

通过分组的键排序——降序：

```
{
  "aggs" : {
    "prices" : {
      "histogram" : {
        "field" : "price",
        "interval" : 50,
        "order" : { "_key" : "desc" }
      }
    }
  }
}
```

通过分组的键升序排序：

```
{
  "aggs" : {
    "prices" : {
      "histogram" : {
        "field" : "price",
        "interval" : 50,
        "order" : { "_count" : "asc" }
      }
    }
  }
}
```

如果直方图聚合有一个直接的指标子聚合，后者可以决定分组的排序方式：

```
{
  "aggs" : {
    "prices" : {
      "histogram" : {
        "field" : "price",
        "interval" : 50,
        "order" : { "price_stats.min" : "asc" }
      },
      "aggs" : { "price_stats" : { "stats" : {} } }
    }
  }
}
```

排序参数 "price_stats.min" : "asc" 表示会根据名为 price_stats 的子聚合里的 min 值对分组进行排序。

不需要为 price_stats 聚合配置价格字段，它会默认从直方图父聚合中继承。

3. 偏移

分组的键默认从 0 开始然后以 interval 的值为间隔步进。我们可以使用 offset 选项来改变分组的范围。

举例说明，有 10 个文档，值的范围是 5 到 14，间隔为 10 的情况下，结果是两个分组各有 5 个文档。如果给出一个额外的偏移量 5，就只有一个分组 [5 - 14] 装着这 10 个文档。

5.3.3 日期直方图聚合

日期直方图聚合是一个多分组聚合，除了只能应用于日期型的值之外，和“直方图聚合”的作用是一样的。

自从 Elasticsearch 将日期类型作为内置类型之后，在日期上进行普通的直方图聚合得到了很好的支持。但是这么做会在精度上有所失真，这是因为时间间隔是不固定的（思考一下，一年或者一个月是多少天）。因为这个原因，我们需要对基于数据的时间进行特别支持。从功能性的角度来看，最主要的不同就是可以通过日期 / 时间表达式指定间隔。

以一个月为间隔请求：

```
{
  "aggs" : {
    "articles_over_time" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      }
    }
  }
}
```

可用的间隔表达式为：year、month、week、day、hour、quarter、minute、second。

1. 时间键值

本质上，日期表示为一个 64 位的时间戳数字，代表从纪元开始到现在的毫秒数。这些时间戳作为分组的键返回。key_as_string 是相同的时间戳转换为格式化的日期字符串，格式通过 format 参数指定。

如果日期格式没有被指定，就会使用字段映射中指定的第一个日期格式。

请求示例：

```
{
  "aggs" : {
    "articles_over_time" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "1M",
        "format" : "yyyy-MM-dd"
      }
    }
  }
}
```

返回值：

```

{
  "aggregations": {
    "articles_over_time": {
      "buckets": [
        {
          "key_as_string": "2013-02-02", // 显示时间
          "key": 1328140800000, // 时间键值
          "doc_count": 1
        },
        {
          "key_as_string": "2013-03-02",
          "key": 1330646400000,
          "doc_count": 2
        },
        ...
      ]
    }
  }
}

```

2. 时间区间

日期时间以世界标准时区 (UTC) 存储在 Elasticsearch 中。默认情况下, 所有的分组和取整的操作也都是以世界标准时完成的。time_zone 参数用于表示分分组时需要使用不同的时间区间。

时间区间也可以指定为一个 ISO 8601 UTC 偏移 (例如 +01:00 或者 -08:00) 或者作为一个互联网编码数据库中的时间区间标识符, 比如 America/Los_Angeles。

```

PUT secisland/log/1
{"date": "2015-10-01T00:30:00Z"}

```

```

PUT secisland/log/2
{"date": "2015-10-01T01:30:00Z"}

```

```

GET secisland/_search?size=0
{
  "aggs": {
    "by_day": {
      "date_histogram": {"field": "date", "interval": "day"}
    }
  }
}

```

如果没有指定时间区间, 就会使用世界标准时间, 结果就是这些文档会放在同一天的分组里, 这个分组开始于 2015 年 10 月 1 日午夜:

```

"aggregations": {
  "by_day": {
    "buckets": [
      {

```

```

    "key_as_string": "2015-10-01T00:00:00.000Z",
    "key": 1443657600000,
    "doc_count": 2
  }
}

```

5.3.3 日期直方图聚合

日期直方图聚合是一个多桶聚合，除了只能用于日期类型的值之外，和“直方图聚合”类似。如果指定一个 -01:00 的时间区间，午夜时间就会比世界标准时间的午夜提前一个小时：请求：GET http://127.0.0.1:9200/secisland/_search?size=0

```

{
  "aggs": {
    "by_day": {
      "date_histogram": {
        "field": "date",
        "interval": "day",
        "time_zone": "-01:00"
      }
    }
  }
}

```

现在，第一个文档就会划入 2015 年 9 月 30 日的分组里，第二个文档划入 2015 年 10 月 1 日的分组里：

```

"aggregations": {
  "by_day": {
    "buckets": [
      {
        "key_as_string": "2015-09-30T00:00:00.000-01:00",
        "key": 1443571200000,
        "doc_count": 1
      },
      {
        "key_as_string": "2015-10-01T00:00:00.000-01:00",
        "key": 1443657600000,
        "doc_count": 1
      }
    ]
  }
}

```

key_as_string 值代表指定的时区中每一天的午夜。

3. 偏移

offset 参数通过指定正 (+) 或负偏移 (-) 时间来修改每个分组的时间起始值，比如 1h 代表一小时，1M 代表一个月。

例如，当用 day 作为时间间隔，每个分组的时间范围是从午夜到午夜。设置 offset 参数为 +6h，会修改每个分组的时间范围为从 6 点到 6 点。

```
PUT secisland/log/1
{"date": "2015-10-01T05:30:00Z"}
```

```
PUT secisland/log/2
{"date": "2015-10-01T06:30:00Z"}
```

```
GET secisland/_search?size=0
```

```
{
  "aggs": {
    "by_day": {
      "date_histogram": {"field": "date", "interval": "day", "offset": "+6h"}
    }
  }
}
```

返回值:

```
"aggregations": {
  "by_day": {
    "buckets": [
      {
        "key_as_string": "2015-09-30T06:00:00.000Z", //10月1日6时前划入了9月30日区间
        "key": 1443592800000,
        "doc_count": 1
      },
      {
        "key_as_string": "2015-10-01T06:00:00.000Z",
        "key": 1443679200000,
        "doc_count": 1
      }
    ]
  }
}
```

开始时间偏移量在时间区间做出调整之后是计算在内的。

5.3.4 时间范围聚合

时间范围聚合是一个专门用于时间型数据的范围聚合。与普通的范围聚合最大的区别在于, `from` 和 `to` 参数值可以使用日期数学表达式, 也可以指定返回的响应中 `from` 和 `to` 字段的日期格式。注意时间范围聚合的每个范围里面包含 `from` 值但排除 `to` 值。

示例:

```
{
  "aggs": {
    "range": {
      "date_range": {
        "field": "date",
        "format": "MM-yyyy",
        "ranges": [
```



```

      { "to": "now-10M/M" },
      { "from": "now-10M/M" }
    ]
  }
}

```

说明：现在减去 10 个月，并跳转到月份的开始时间。

在上面的例子里，我们创建了两个范围分组，第一个会包含日期在 10 个月之前的所有文件，第二个会包含从 10 个月前到现在的所有文件。

返回值：

```

{
  ...
  "aggregations": {
    "range": {
      "buckets": [
        { //10 月内的数据
          "to": 1.3437792E+12,
          "to_as_string": "08-2012",
          "doc_count": 7
        },
        { // 近 10 个月前的数据
          "from": 1.3437792E+12,
          "from_as_string": "08-2012",
          "doc_count": 2
        }
      ]
    }
  }
}

```

5.3.5 范围聚合

范围聚合是一个基于多组值来源的聚合，可以让用户定义一系列范围，每个范围代表一个分组。在聚合执行的过程中，从每个文档提取出来的值都会检查每个分组的范围，并且使相关的文档落入分组中。注意，范围聚合的每个范围内包含 from 值但是排除 to 值。

示例：

```

{
  "aggs" : {
    "price_ranges" : {
      "range" : {
        "field" : "price",
        "ranges" : [
          { "to" : 50 },
          { "from" : 50, "to" : 100 },

```

```

    { "from" : 100 }
  ]
}
}
}

```

返回值:

```

{
  "aggregations": {
    "price_ranges": {
      "buckets": [
        { // 查询中第一个范围
          "to": 50,
          "doc_count": 2
        },
        { // 查询中第二个范围
          "from": 50,
          "to": 100,
          "doc_count": 4
        },
        { // 查询中第三个范围
          "from": 100,
          "doc_count": 4
        }
      ]
    }
  }
}

```

5.3.6 过滤聚合

过滤聚合是一个单分组聚合，包含当前文档集中所有匹配指定的过滤条件的文档。通常用于从当前聚合中得到一个具体的文档集。

示例:

```

{
  "aggs" : {
    "red_products" : {
      "filter" : { "term": { "color": "red" } },
      "aggs" : { "avg_price" : { "avg" : { "field" : "price" } } }
    }
  }
}

```

通过这个例子，我们可以计算出所有红色产品的平均价格。

响应:

```
{
  "aggs" : {
    "red_products" : { "doc_count" : 100, "avg_price" : { "value" : 56.3 }}
  }
}
```

5.3.7 多重过滤聚合

多重过滤聚合是定义一个多分组聚合，每个分组关联一个过滤条件，并收集所有满足自身过滤条件的文档。

示例：

```
{
  "aggs" : {
    "messages" : {
      "filters" : {
        "filters" : {
          "errors" : { "term" : { "body" : "error" } },
          "warnings" : { "term" : { "body" : "warning" } }
        }
      },
      "aggs" : {
        "monthly" : {
          "histogram" : {
            "field" : "timestamp",
            "interval" : "1M"
          }
        }
      }
    }
  }
}
```

在这个例子里，我们分析日志信息。聚合会创建两个关于日志数据的分组，一个收集包含错误信息的文档，另一个收集包含告警信息的文档。而且每个分组会按月份划分。

返回值：

```
...
"aggs" : {
  "messages" : {
    "buckets" : {
      "errors" : { // 错误的结果
        "doc_count" : 34,
        "monthly" : {
          "buckets" : [
            ...
          ]
        }
      }
    }
  }
}
```

```

    },
    "warnings" : {      // 告警的结果
      "doc_count" : 439,
      "monthly" : {
        "buckets" : [

```

1. 匿名过滤

过滤字段可以作为过滤数组提供，经过过滤生成的分组会以请求的顺序返回。

2. 其他分组

设置 `other_bucket` 参数可以在响应中添加一个分组，用来收集所有没有匹配到任何给出的过滤条件的文档：

❑ `false`：表示不计算其他分组。

❑ `true`：返回 `other` 分组，如果是命名过滤聚合，这个分组会被默认命名为 `_other_`；如果是匿名过滤聚合，这个分组会作为最后一个返回。

`other_bucket_key` 参数可以修改其他分组的默认键 `_other_`。设置这个参数就代表 `other_bucket` 参数值为 `true`。

示例展示请求一个名为 `other_messages` 的其他分组：

```

{
  "aggs" : {
    "messages" : {
      "filters" : {
        "other_bucket_key": "other_messages",
        "filters" : {
          "errors" : { "term" : { "body" : "error" } },
          "warnings" : { "term" : { "body" : "warning" } }
        }
      }
    },
    "aggs" : {
      "monthly" : {
        "histogram" : {
          "field" : "timestamp",
          "interval" : "1M"
        }
      }
    }
  }
}

```

```

    }
  }
}

```

返回值:

```

...
"aggs" : {
  "messages" : {
    "buckets" : {
      "errors" : {
        ...
      },
      "warnings" : {
        ...
      },
      "other_messages" : { // 其他分组结果
        "doc_count" : 237,
        "monthly" : {
          "buckets" : [
            ...
          ]
        }
      }
    }
  }
}
...

```

5.3.8 空值聚合

空值聚合是一个基于字段数据的单分组聚合，在当前文档集中对所有缺失字段值（字段值为空或者被设置为 null）的文档创建一个分组。这个聚合通常和其他字段数据分组聚合（例如范围聚合）一起使用，返回由于缺少字段值而不能放入其他任何分组中的所有文档的信息。

示例:

```

{
  "aggs" : {
    "products_without_a_price" : { "missing" : { "field" : "price" } }
  }
}

```

在这个例子中，我们获取所有没有价格的产品总量。

返回值:

```

{
  ...
  "aggs" : {
    "products_without_a_price" : {
      "doc_count" : 10
    }
  }
}

```



```

    }
  }
}

```

5.3.9 嵌套聚合

嵌套聚合是一种特殊的单分组聚合，可以聚合嵌套的文档。

例如，假设我们有一个产品索引，每个产品来自不同的经销商——有着不同的价格。索引的映射如下：

```

{
  ...
  "product" : {
    "properties" : {
      "resellers" : {
        "type" : "nested",
        "properties" : {
          "name" : { "type" : "string" },
          "price" : { "type" : "double" }
        }
      }
    }
  }
}

```

product 对象下面的 resellers 是一个包含嵌套文档的数组：

```

{
  "query" : { "match" : { "name" : "led tv" } },
  "aggs" : {
    "resellers" : {
      "nested" : { "path" : "resellers" },
      "aggs" : { "min_price" : { "min" : { "field" : "resellers.price" } } }
    }
  }
}

```

这个示例会返回可以买到的产品 (led tv) 的最低价。

嵌入聚合需要定义嵌入文档在顶级文档中的路径 (path)。我们可以在这些嵌入文档上定义任何类型的聚合。

返回值：

```

{
  "aggregations": {
    "resellers": {
      "min_price": { "value" : 350 }
    }
  }
}

```

5.3.10 采样聚合



警告 这个功能是实验性的，可能会在未来的版本中被修改或者移除。

采样聚合是一个过滤聚合，用在子聚合来限制得分最高的文档的样本。不同的设置可以用于限制共享一个共同值匹配的数量，如“作者”。

使用环境：

- ❑ 重点分析高相关性的匹配而不是包含一大堆低质量的匹配。
- ❑ 移除分析中的偏差来确保从不同来源获取内容的公平性。
- ❑ 为了减少聚合的运行成本，可能通过 `significant_terms` 只处理有用的结果。

示例：

```
{
  "query": {
    "match": {"text": "iphone"}
  },
  "aggs": {
    "sample": {
      "sampler": {
        "shard_size": 200,
        "field": "user.id"
      },
      "aggs": {
        "keywords": {
          "significant_terms": {
            "field": "text"
          }
        }
      }
    }
  }
}
```

返回值：

```
{
  ...
  "aggregations": {
    "sample": {
      "doc_count": 1000, // 总共 1000 个文档
      "keywords": { // significant_terms 结果
        "doc_count": 1000,
        "buckets": [
          ...
          {
            "key": "bend",
```

```

      "doc_count": 58,
      "score": 37.982536582524276,
      "bg_count": 103
    },
    ....
  }
}

```

总共抽取出 1000 个文档是因为我们的索引有 5 个分片，每个分片返回 200 个。

例如我们询问在采样中的用户中哪个用户的发消息最多，`significant_terms` 聚集的结果是会被任何单一过度活跃的用户影响的。

1. 分片规模

`shard_size` 参数限制每个分片执行采样的过程中多少高得分文档会被采集。默认值是 100。

2. 控制差异

可以任意使用字段或者脚本以及 `max_doc_per_value` 来控制任何一个分片采集具有相同值的文档的最大数量。值的选择（例如 `author`）来自于一个固定的字段或者由一个脚本动态派生。

如果选择的字段或者脚本为一个文档生成了多个值，聚合会报错。出于效率方面的考虑，目前不提供重复使用许多值的形式。

提示 当采样数据的时候，应该确保样本代表大量的不同情况，而不要被单一的情况影响样本。聚合也是一样，通过多种设置进行采样可以提供一种方法来消除内容中的偏差（例如一个人口稠密的地理位置，一个时间线上的尖峰，一个过于活跃的垃圾论坛）。

3. 字段

利用字段控制差异。

```

{
  "aggs" : {
    "sample" : {
      "sampler" : {
        "field" : "author",
        "max_docs_per_value" : 3
      }
    }
  }
}

```

注意 `max_docs_per_value` 设置只为了分片局部采样的目的应用在每个分片的基础上。不作为在整体搜索结果上提供去重功能的方法。

4. 脚本

利用脚本控制差异:

```
{
  "aggs" : {
    "sample" : {
      "sampler" : { "script" : "doc['author'].value + '/' + doc['genre'].value" }
    }
  }
}
```

注意在这个例子中,我们选择使用默认的 `max_docs_per_value` 设置 1 以及结合 `author` 和 `genre` 字段来确保每个分片采集的样本最多有一个匹配 `author/genre` 对。

5.3.11 重要索引词聚合

重要索引词聚合返回一组索引词中令人关注或者不同寻常的事件。



警告 当聚合在大的索引中运行的时候,会变得非常笨重。Elasticsearch 的团队正致力于提供一种更轻量级的抽样技术。这意味着,未来会以一种不向后兼容的方式改变这个接口。

使用示例:

- 当用户在文本中搜索“禽流感”的时候,给出建议“H5N1”。
- 从信用卡的交易历史报告中,可以分析持卡人损失,从而确认商户的共同特性。
- 建议从股票代码的关键词中实现新闻分类自动化。
- 从不诚信的医生诊断报告中发现他们更多的分享颈伤的诊断。
- 指出爆胎率超标的轮胎厂商。

在刚才的这些示例中发现,我们要搜索的索引词并不是简单的关键词的排行,而是在一段时间内,经历了重大改变的索引词。例如在索引的 1000 万个文档中仅存在 5 个文档包含索引词“H5N1”,而现在索引 100 个文档中有 4 个文档包含“H5N1”。5/10000000 和 4/100 相比,在频率上明显不同,改进很大。

1. 单一分析

最简单的情况下,前台集是匹配查询条件的搜索结果,用于统计对比的后台集是聚合结果的索引。

示例:

```
{
  "query" : { "terms" : { "force" : [ "British Transport Police" ] } },
  "aggregations" : {
```

```

    "significantCrimeTypes" : { "significant_terms" : { "field" : "crime_type" } }
  }
}

```

返回值:

```

{
  ...
  "aggregations" : {
    "significantCrimeTypes" : {
      "doc_count": 47347,
      "buckets" : [
        {
          "key": "Bicycle theft",
          "doc_count": 3640,
          "score": 0.371235374214817,
          "bg_count": 66799
        }
      ]
    }
  }
}

```

当查询所有的警察机关的犯罪指数时, 这些结果显示英国交警处理的自行车盗窃案的比例不符合预期。一般情况下, 自行车盗窃的犯罪率仅有 1% (66799/5064554), 但英国交警抓到的 7% (3640/47347) 的罪犯都是自行车盗窃犯。这显然是七倍的频率, 所以将自行车盗窃案强调为最主要的犯罪类型。

上面的例子是用来寻找异常的犯罪类型, 它只查询了英国交警的集合作为比较的数据集。为了寻找所有警察机构的异常犯罪类型, 我们需要为每一个不同的机构进行重复查询。用这种方式寻找异常是非常烦琐的。

2. 多重分析

这是一个用来执行跨多个类别分析的简单方法, 利用父级聚合来划分数据。

使用父聚合来划分数据的例子:

```

{
  "aggregations": {
    "forces": {
      "terms": { "field": "force" },
      "aggregations": {
        "significantCrimeTypes": {
          "significant_terms": { "field": "crime_type" }
        }
      }
    }
  }
}

```


返回值:

```
{
  ...
  "aggregations": {
    "forces": {
      "buckets": [
        {
          "key": "Metropolitan Police Service", // 伦敦警察厅
          "doc_count": 894038,
          "significantCrimeTypes": {
            "doc_count": 894038,
            "buckets": [
              {
                "key": "Robbery",
                "doc_count": 27617,
                "score": 0.0599,
                "bg_count": 53182
              },
              ...
            ]
          }
        },
        {
          "key": "British Transport Police", // 英国交通警察
          "doc_count": 47347,
          "significantCrimeTypes": {
            "doc_count": 47347,
            "buckets": [
              {
                "key": "Bicycle theft",
                "doc_count": 3640,
                "score": 0.371,
                "bg_count": 66799
              },
              ...
            ]
          }
        },
        ...
      ]
    }
  }
}
```

现在, 我们使用一条请求就对每个警察机构做了异常检查。

我们可以使用其他形式的聚合来划分我们的数据, 例如对地理区域进行划分, 来分析异常犯罪类型的地址集团分布。

```
{
  "aggs": {
    "hotspots": {
      "geohash_grid": {
```

```

    "field": "location",
    "precision": 5,
  },
  "aggs": {
    "significantCrimeTypes": {
      "significant_terms": { "field": "crime_type" }
    }
  }
}

```

这个示例利用地理网格聚合来创建基于地理位置的分组结果，我们也可以根据每个分组的区域定义一个犯罪类型的异常级别。

5.3.12 索引词聚合

索引词聚合是一个基于聚合产生一个多组的值，每个分组都是由一个独特的值动态创建。

以下示例是根据采集类型进行聚合：

```

{
  "aggs": {
    "terms": {
      "terms": {
        "field": "collectType"
      }
    }
  }
}

```

返回值：

```

"aggregations": {
  "terms": {
    "doc_count_error_upper_bound": 0,
    "sum_other_doc_count": 0,
    "buckets": [
      {
        "key": "web-log",
        "doc_count": 51294
      },
      {
        "key": "monitor",
        "doc_count": 4990
      },
      {
        "key": "linux",
        "doc_count": 440
      },

```

```

    ]
  }
}

```

`doc_count_error_upper_bound` 代表每个索引词文档计数的错误上限。

当存在大量的不同索引词, Elasticsearch 只返回排名靠前的索引词。

`sum_other_doc_count` 代表所有没被放在响应中的分组里的文档计数之和。

默认情况下, 索引词通过文档计数 (`doc_count`) 排序, 条件聚合会返回排名前十的索引词生成的分组。我们可以通过设置 `size` 参数对返回结果的数量进行修改。

1. 规模

通过设置 `size` 参数, 可以定义需要返回的索引词分组的数量。默认情况下, 协调搜索进程的节点会请求每个分片提供顶层与规模大小相同的索引词分组。当所有分片响应完成时, 节点会缩小最终结果的规模并返回给用户。

这意味着如果不同索引词的数量大于规模, 返回的列表会发生偏移和失真 (可能是索引词计数偏移, 也可能是本应出现的索引词分组没有被返回)。

如果规模设置为 0, 规模的大小将会是 `Integer.MAX_VALUE`。

2. 排序

通过设置 `order` 参数, 可以用自定义分组的排序方式。默认情况下, 分组根据其 `doc_count` 参数降序排序。也可以通过以下方式改变这种行为:

根据分组的 `doc_count` 升序排序:

```

{
  "aggs" : {
    "genders" : {
      "terms" : {
        "field" : "gender",
        "order" : { "_count" : "asc" }
      }
    }
  }
}

```

根据分组的索引词的字母顺序进行升序排序:

```

{
  "aggs" : {
    "genders" : {
      "terms" : {
        "field" : "gender",
        "order" : { "_term" : "asc" }
      }
    }
  }
}

```

通过单值指标子聚合排序（通过聚合名称进行定义）：

```
{
  "aggs" : {
    "genders" : {
      "terms" : {
        "field" : "gender",
        "order" : { "avg_height" : "desc" }
      },
      "aggs" : { "avg_height" : { "avg" : { "field" : "height" } } }
    }
  }
}
```

通过多值指标子聚合进行排序（通过聚合名称进行定义）：

```
{
  "aggs" : {
    "genders" : {
      "terms" : {
        "field" : "gender",
        "order" : { "height_stats.avg" : "desc" }
      },
      "aggs" : { "height_stats" : { "stats" : { "field" : "height" } } }
    }
  }
}
```

无论聚合路径上最后一个聚合是单分组还是多组，只要聚合路径是一个单分组类型，就可以基于更“深”一层的聚合对分组进行排序。单分组类型排序会基于分组里文档的数字（如 doc_count）进行；多组类型应用相同的规则（多组度量聚合中，路径必须标明用来排序的指标名称；如果是单组度量聚合，得到的值将用于排序）。

路径的定义方式如下所示：

- 聚合分隔符为 >
- 指标分隔符为 .
- 聚合名为 < 聚合的名称 >
- 指标为 < 指标的名称（如果是多值度量聚合）>
- 路径为 < 聚合名 >[< 聚合分隔符 >< 聚合名 >]*[< 指标分隔符 >< 指标 >]

```
{
  "aggs" : {
    "countries" : {
      "terms" : {
        "field" : "address.country",
        "order" : { "females>height_stats.avg" : "desc" }
      },
      "aggs" : {
        "females" : {
```

```

    "filter" : { "term" : { "gender" : "female" } },
    "aggs" : {
      "height_stats" : { "stats" : { "field" : "height" } }
    }
  }
}

```

这个示例是基于女性人群的平均身高对国家这个指标进行排序。

可以通过提供排序标准的数组，利用多个排序标准对分组进行排序，下例在先前的基础上添加了基于 doc_count 的降序排序。

```

{
  "aggs" : {
    "countries" : {
      "terms" : {
        "field" : "address.country",
        "order" : [ { "females>height_stats.avg" : "desc" }, { "_count" : "desc" } ]
      },
      "aggs" : {
        "females" : {
          "filter" : { "term" : { "gender" : { "female" } } },
          "aggs" : { "height_stats" : { "stats" : { "field" : "height" } } }
        }
      }
    }
  }
}

```

如果两个分组在所有的排序标准下拥有相同的值，将用分组的索引词进行首字母升序排序，确保分组排序的一致性。

3. 过滤值

可以利用 include 和 exclude 字段对创建分组的索引词进行过滤，这个过程基于正则表达式的字符串或者一组精确值。如下所示：

```

{
  "aggs" : {
    "tags" : {
      "terms" : {
        "field" : "tags",
        "include" : ["mazda", "honda"],
        "exclude" : "water_.*"
      }
    }
  }
}

```


include 参数决定哪些值“允许”被聚合，exclude 参数决定哪些值不能被聚合。当同时定义两个参数时，exclude 具有优先权，这意味着 include 将会在 exclude 生效之前发挥作用。因此，同时包含 include 和 exclude 参数值的索引词将会被忽略。

语法和正则表达式查询的语法相同。

基于精准值的匹配，include 和 exclude 参数可以简单地对一个字符串数组进行赋值，表示在索引中被查找的索引词：

```
{
  "aggs" : {
    "JapaneseCars" : {
      "terms" : {
        "field" : "make",
        "include" : ["mazda", "honda"]
      }
    },
    "ActiveCarManufacturers" : {
      "terms" : {
        "field" : "make",
        "exclude" : ["rover", "jensen"]
      }
    }
  }
}
```

4. 多字段索引词聚合

索引词聚合不支持在同一类型文档的多个字段中采集索引词。原因是索引词聚合不采集字符串索引词自身的值，而是利用全局序数来生成一个字段中所有唯一值的列表。全局序数可以带来重要的性能提升，所以不能跨多个字段。

有以下两个方式可用于在多个字段上执行索引词聚合：

- ❑ 脚本——利用脚本在多个字段上重复进行索引词聚合，但是这会使全局序数的优化失效而且会比从一个字段上采集索引词更慢。
- ❑ copy_to 字段——如果事先知道你想要从哪些字段中采集索引词，就可以在映射中利用 copy_to 来创建一个专用字段包含这些字段的值。可以在这一个字段上执行聚合，并享受全局序数的优化效果。

5. 采集模式

这种模式为推迟子聚合的计算。当字段拥有很多不同的索引词但必要的结果却很少的时候，直到上层的父聚合精简之后再执行子聚合的计算会更有效率。通常，聚合树的分支会优先在深度上扩展，然后才会进行精简。在一些特殊的情况下，这会非常浪费系统资源而且会受到内存的限制。

举个例子，查询一个电影数据库，找出 10 个最受欢迎的演员和他们最常合作的 5 个演

员，代码如下：


```
{
  "aggs" : {
    "actors" : {
      "terms" : {
        "field" : "actors",
        "size" : 10
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" : 5
          }
        }
      }
    }
  }
}
```

虽然电影的数量可能相对较小而且我们要的仅仅是 50 个结果分组，但在计算的过程中，分组的数量可能会相当大——每个电影会产生 n^2 个分组，其中， n 代表演员的数量。明智的选择是首先确定 10 个最受欢迎的演员，然后再检查这 10 个演员最常合作的演员。这个策略就是我们所说的与深度优先模式相对的广度优先采集模式，如下所示：

```
{
  "aggs" : {
    "actors" : {
      "terms" : {
        "field" : "actors",
        "size" : 10,
        "collect_mode" : "breadth_first"
      },
      "aggs" : {
        "costars" : {
          "terms" : {
            "field" : "actors",
            "size" : 5
          }
        }
      }
    }
  }
}
```


当使用广度优先模式时，这些落入最上分组里的文档会被缓存以便随后复用。所以会有一个与匹配的文档数量线性相关的内存开销。在大多数请求中，生成的分组体积比落入的文档数量要小，所以默认的深度优先采集模式通常是最好的选择。但有些情况下，广度优先策

略会效率显著。现在 Elasticsearch 默认使用深度优先采集模式，除非显式指出使用广度优先模式。

 **警告** 使用广度优先采集模式的时候，不可以在聚合中嵌入如 `top_hits` 这种需要访问匹配得分信息的聚合。这是因为这么做需要缓存每个文档的浮点得分值，会造成非常昂贵的内存开销。

5.3.13 总体聚合

总体聚合是在搜索执行的环境中定义一个包含所有文档的单分组。环境被搜索的索引和文档类型定义，不受搜索查询本身的影响。

 **注意** 总体聚合只能作为顶级聚合使用（把总体聚合嵌入其他分组聚合中是没有意义的）。

示例如下：

```
{
  "query" : { "match" : { "title" : "shirt" } },
  "aggs" : {
    "all_products" : {
      "global" : {},
      "aggs" : { "avg_price" : { "avg" : { "field" : "price" } } }
    }
  }
}
```

总体聚合的聚合体是空的。这个示例展示了如何忽略查询条件，在搜索环境中的所有文档上计算聚合。

返回值如下：

```
{
  ...
  "aggregations" : {
    "all_products" : {
      "doc_count" : 100,
      "avg_price" : { "value" : 56.3 }
    }
  }
}
```

5.3.14 地理点距离聚合

作用于地理点类型字段上面的多组聚合，原理和范围聚合非常像。用户可以定义一个原

点和一系列距离范围分组。聚合评估每个文档的值和原点之间的距离，然后基于范围决定文档属于哪一个分组（一个文档仅属于一个分组，文档和原点的距离在分组的距离范围内），示例如下：

```
{
  "aggs": {
    "rings_around_amsterdam": {
      "geo_distance": {
        "field": "location",
        "origin": "52.3760, 4.894",
        "ranges": [
          { "to": 100 },
          { "from": 100, "to": 300 },
          { "from": 300 }
        ]
      }
    }
  }
}
```

返回值如下：

```
{
  "aggregations": {
    "rings": {
      "buckets": [
        {
          "key": "*-100.0",
          "from": 0,
          "to": 100.0,
          "doc_count": 3
        },
        {
          "key": "100.0-300.0",
          "from": 100.0,
          "to": 300.0,
          "doc_count": 1
        },
        {
          "key": "300.0-*",
          "from": 300.0,
          "doc_count": 7
        }
      ]
    }
  }
}
```

所指定的字段必须是地理点类型（只能在映射中明确地设置）。也可以定义一个地理点类型的字段数组，在聚合期间会考虑所有的字段。原点参数可以接受所有地理点类型支持的

格式:

- 对象格式: { "lat": 52.3760, "lon": 4.894 }——这是最安全的格式,因为它最明确地指出了纬度和经度的值。
- 字符串格式: "52.3760, 4.894"——第一个数字代表纬度,第二个数字代表经度。
- 数组格式: [4.894, 52.3760]——基于 GeoJSON 标准,第一个数字代表经度,第二个数字代表纬度。

默认情况下,距离单位是米 (metres),但也可以接受其他距离单位:英里 (miles)、英寸 (inches)、码 (yards)、千米 (kilometers)、厘米 (centimeters)、毫米 (millimeters)。

```
{
  "aggs" : {
    "rings" : {
      "geo_distance" : {
        "field" : "location",
        "origin" : "52.3760, 4.894",
        "unit" : "mi",
        "ranges" : [
          { "to" : 100 },
          { "from" : 100, "to" : 300 },
          { "from" : 300 }
        ]
      }
    }
  }
}
```


有三种距离计算模式: sloppy_arc (默认)、arc (最精确) 和 plane (最快)。可以使用 distance_type 参数进行设置。

5.3.15 地理散列网格聚合

作用于地理点类型字段上的多组聚合,将各个点分配到代表网格中各个单元格的分组里。聚合结果网格可能很稀疏,而且只包含匹配到数据的单元格。每个单元格利用自定义精度下产生的地理散列值进行标记。有如下特点:

- 高精度地理散列字符串长度很长而且每个网格覆盖很小的地理区域。
- 低精度地理散列字符串长度比较短而且每个网格覆盖很大的地理区域。

这个聚合使用的地理散列有精度选择,1 到 12,默认精度是 5。

 以最高的地理散列精度 12 创建网格,覆盖的土地面积要小于一平方米。且如此高的精度需要大量的内存开销,结果的体积也会非常大。

所指定的字段必须是地理点类型 (只能在映射中明确地设置)。也可以定义一个地理点

类型的字段数组，在聚合期间将考虑所有的字段。

1. 简单的低精度请求

```
{
  "aggregations" : {
    "myLarge-GrainGeoHashGrid" : {
      "geohash_grid" : {
        "field" : "location",
        "precision" : 3
      }
    }
  }
}
```

返回值如下：

```
{
  "aggregations": {
    "myLarge-GrainGeoHashGrid": {
      "buckets": [
        { "key": "svz", "doc_count": 10964 },
        { "key": "sv8", "doc_count": 3198 }
      ]
    }
  }
}
```

2. 高精度请求

当请求详细内容的分组（在地图中通常表示为“放大”），一个类似于地理边界聚合的过滤可以用于缩小主题区域。否则，可能会创建和返回上百万的分组。

```
{
  "aggregations" : {
    "zoomedInView" : {
      "filter" : {
        "geo_bounding_box" : {
          "location" : {
            "top_left" : "51.73, 0.9",
            "bottom_right" : "51.55, 1.1"
          }
        }
      },
      "aggregations": {
        "zoom1": {
          "geohash_grid" : {
            "field": "location",
            "precision": 8,
          }
        }
      }
    }
  }
}
```

3. 在赤道上单元格的面积

表 5-1 展示了在各种地理点散列字符串长度下, 各个单元格覆盖的面积指标。单元格面积随着纬度的不同而变化, 所以表格展示的是在赤道上最坏 (面积最大) 的情况。

表 5-1 地理点散列字符串长度对应赤道区域面积

地理点散列字符串长度	区域 (宽 × 高)
1	5 009.4km × 4 992.6km
2	1 252.3km × 624.1km
3	156.5km × 156km
4	39.1km × 19.5km
5	4.9km × 4.9km
6	1.2km × 609.4m
7	152.9m × 152.4m
8	38.2m × 19m
9	4.8m × 4.8m
10	1.2m × 59.5cm
11	14.9cm × 14.9cm
12	3.7cm × 1.9cm

5.3.16 IPv4 范围聚合

就像专用的日期范围聚合一样, IPv4 范围聚合专用于 IPv4 类型字段。

示例如下:

```
{
  "aggs" : {
    "ip_ranges" : {
      "ip_range" : {
        "field" : "ip",
        "ranges" : [
          { "to" : "10.0.0.5" },
          { "from" : "10.0.0.5" }
        ]
      }
    }
  }
}
```

返回值如下:

类型{ 字段名, 在聚合时同时考虑所有的字段。

```
...
"aggregations": {
  "ip_ranges": {
    "buckets" : [
      {
        "to": 167772165,
        "to_as_string": "10.0.0.5",
        "doc_count": 4
      },
      {
        "from": 167772165,
        "from_as_string": "10.0.0.5",
        "doc_count": 6
      }
    ]
  }
}
```

IP 范围也可以用 CIDR 掩码来定义:

```
{
  "aggs" : {
    "ip_ranges" : {
      "ip_range" : {
        "field" : "ip",
        "ranges" : [
          { "mask" : "10.0.0.0/25" },
          { "mask" : "10.0.0.127/25" }
        ]
      }
    }
  }
}
```

返回值如下:


```
{
  "aggregations": {
    "ip_ranges": {
      "buckets": [
        {
          "key": "10.0.0.0/25",
          "from": 1.6777216E+8,
          "from_as_string": "10.0.0.0",
          "to": 167772287,
          "to_as_string": "10.0.0.127",
          "doc_count": 127
        },
        {
          "key": "10.0.0.127/25",
```

```

"from": 1.6777216E+8,
"from_as_string": "10.0.0.0",
"to": 167772287,
"to_as_string": "10.0.0.127",
"doc_count": 127
}
]
}
}
}

```

5.4 管道聚合


 **警告** 这一章功能都是实验性质的，在将来的版本中可能会被修改或删除。

管道聚合工作于其他聚合产生的输出结果而不是文档集，用于向输出树添加信息。有不同类型的管道聚合，每一种从其他聚合中计算不同的信息，这些类型分为如下两大类：

- 父类聚合——在父聚合输出结果的基础上进行管道聚合，可以在现有分组的基础上计算新的分组或者聚合。
- 兄弟聚合——在兄弟聚合输出结果的基础上进行管道聚合，可以计算与兄弟聚合相同等级的新聚合。

通过 `bucket_path` 参数指定请求指标的路径，管道聚合可以引用需要的聚合来执行计算。

管道聚合不能拥有子聚合，但是可以在 `buckets_path` 参数中引入另一个管道聚合，使管道聚合链接起来。

 **注意** 因为管道聚合仅在输出结果中添加信息，当链接管道聚合的时候，每个管道聚合的输出结果都会被包含在最终结果里。

1. `buckets_path` 语法

大多数管道聚合需要另一个聚合作为它的输入。输入聚合通过 `buckets_path` 参数定义，格式如下：

- 聚合分隔符为 >
- 指标分隔符为 .
- 聚合名为 <聚合的名称>
- 指标为 <指标的名称（如果是多值度量聚合）>
- 路径为 <聚合名>[<聚合分隔符><聚合名>]*[<指标分隔符><指标>]

举个例子，路径“my_bucket>my_stats.avg”会将“my_stats”中的平均值指标作为“my_bucket”分组聚合的输入。

路径跟管道聚合的位置是相对的；并不是绝对路径，而且路径不能沿着聚合树“向上”返回。例如，在日期直方图聚合中嵌入移动平均值聚合，并指出“兄弟聚合”指标“the_sum”：

```
{
  "my_date_histo": {
    "date_histogram": {
      "field": "timestamp",
      "interval": "day"
    },
    "aggs": {
      "the_sum": { "sum": { "field": "lemmings" } },
      "the_movavg": { "moving_avg": { "buckets_path": "the_sum" } }
    }
  }
}
```

buckets_path 通过一个相对路径“the_sum”指出聚合指标。

buckets_path 也用于兄弟管道聚合，管道聚合是“靠近”一系列分组聚合而不是嵌入它们“里面”。

例如，max_bucket 聚合利用 buckets_path 指定了一个嵌入兄弟聚合的指标：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "sales": {
          "sum": {
            "field": "price"
          }
        }
      }
    },
    "max_monthly_sales": {
      "max_bucket": {
        "buckets_path": "sales_per_month>sales"
      }
    }
  }
}
```


2. 特殊路径

与指标路径不同, `buckets_path` 可以使用特殊的 “`_count`” 路径, 让管道聚合使用文档计数作为输入参数。例如, 移动平均值聚合可以计算每个分组的文档计数而不是给出具体的指标, 示例如下:

```
{
  "my_date_histo": {
    "date_histogram": {
      "field": "timestamp",
      "interval": "day"
    },
    "aggs": {
      "the_movavg": { "moving_avg": { "buckets_path": "_count" } }
    }
  }
}
```

3. 处理空数据

现实世界中的数据通常是嘈杂的, 有时候还会存在空数据——数据根本不存在。这可能有各种各样的原因, 最常见的原因是:

- ❑ 落入分组中的文档不包含请求的字段。
- ❑ 一个或多个分组没有匹配到文档。
- ❑ 可能因为其他依赖的分组缺失了数据, 被计算的指标无法产生值。

缺口策略是一个用来通知管道聚合当遭遇到数据缺口时应该执行哪种行为的机制。所有的管道聚合都支持 `gap_policy` 参数指定缺口策略。现在有两种缺口策略可以选择:

- ❑ 跳过 (skip) —— 这个选项将缺失数据的分组视为不存在, 会跳过这个分组并使用下一个可用值继续计算。
- ❑ 插入零值 (insert_zeros) —— 这个选项会用零替代缺失值, 而且管道聚合会照常进行计算。

5.4.1 平均分组聚合

平均分组聚合会计算在一组聚合中指定指标的平均值。指定的指标必须是数字型而且这个组聚合必须是多组聚合。用 `avg_bucket` 来表示平均分组聚合。

组聚合结构如下:

```
{
  "avg_bucket": { "buckets_path": "the_sum" }
}
```

参数如下所示:

- ❑ `buckets_path` —— 想要计算平均值的分组路径 (必要参数)。
- ❑ `gap_policy` —— 当数据缺口出现时应用的策略 (默认为跳过)。

□ format——用于规范聚合输出值的格式（默认为 null）。

计算每月销售总额的平均值的示例如下：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "sales": {
          "sum": { "field": "price" }
        }
      }
    },
    "avg_monthly_sales": {
      "avg_bucket": { "buckets_path": "sales_per_month>sales" }
    }
  }
}
```

返回值如下：

```
{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {
            "value": 550 //2015 年 1 月销售总额
          }
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {
            "value": 60 //2015 年 2 月销售总额
          }
        }
      ],
      "avg_monthly_sales": { //1 月和 2 月的平均值
        "value": 305
      }
    }
  }
}
```

5.4.2 移动平均聚合

给出一组有序的数据，移动平均聚合会在数据上滑动一个固定大小的窗口并且给出窗口的平均值。例如，给出的数据是 [1,2,3,4,5,6,7]，我们可以计算一个窗口大小为 5 的简单移动平均值聚合：

```
(1+2+3+4+5) / 5=3
(2+3+4+5+6) / 5=4
(3+4+5+6+7) / 5=5
```

移动平均值是对数据序列进行平滑处理的简单方法。通常用于基于时间的数据，比如股票价格或者服务指标。平滑可以用来消除高频波动或随机噪声，使低频趋势变得更加容易发现。

用 moving_avg 来表示移动平均聚合，其结构如下：

```
{
  "moving_avg": {
    "buckets_path": "the_sum",
    "model": "holt",
    "window": 5,
    "gap_policy": "insert_zero",
    "settings": {"alpha": 0.8}
  }
}
```

参数如下所示：

- buckets_path——需要的指标路径（必要参数）。
- model——我们想用的移动平均加权模型。
- gap_policy——决定遭遇数据缺口时的行为（默认插入零值）。
- window——在直方图上“滑动”的窗口大小（默认为 5）。
- settings——模型的具体设置，根据指定的模型有不同的内容。

注意，移动平均聚合必须嵌入到一个直方图或者日期直方图聚合中使用。嵌入方式同其他度量聚合相同：

```
{
  "my_date_histo": {
    "date_histogram": {
      "field": "timestamp",
      "interval": "day"
    },
    "aggs": {
      "the_sum": { "sum": { "field": "lemmings" } },
      "the_movavg": { "moving_avg": { "buckets_path": "the_sum" } }
    }
  }
}
```

可以在直方图聚合里随意加入普通的度量聚合，最后把移动平均聚合嵌入到直方图中。然后用 `buckets_path` 参数“指出”一个直方图聚合中的兄弟度量聚合。

5.4.3 总和和分组聚合

总和和分组聚合用于计算一组聚合创建的所有分组中指定指标的和。指定的指标必须是数字型而且这个组聚合必须是多组聚合。

用 `sum_bucket` 来表示总和和分组聚合，其结构如下：

```
{
  "sum_bucket": {"buckets_path": "the_sum"}
}
```

参数见 5.4.1 节。

计算所有月销售总额 `sales` 分组的和如下所示：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {"field": "date", "interval": "month"},
      "aggs": {
        "sales": {
          "sum": {"field": "price"}
        }
      }
    },
    "sum_monthly_sales": {
      "sum_bucket": {"buckets_path": "sales_per_month>sales"}
    }
  }
}
```

5.4.4 总和累计聚合

总和累计聚合父管道聚合，计算父直方图（或日期直方图）聚合中指定的指标的累计值。指定的指标必须是数字型而且直方图聚合必须设置 `min_doc_count` 为 0。

用 `cumulative_sum` 来表示总和和累计聚合，其结构如下：

```
{
  "cumulative_sum": {"buckets_path": "the_sum"}
}
```

参数可参见 5.4.1 节

计算所有月销售总额 `sales` 分组的累计和如下所示：

```
{
  "aggs" : {
    "sales_per_month" : {
```

```

    "date_histogram" : {
      "field" : "date",
      "interval" : "month"
    },
    "aggs": {
      "sales": {
        "sum": {"field": "price"}
      },
      "cumulative_sales": {
        "cumulative_sum": {
          "buckets_path": "sales"
        }
      }
    }
  }
}

```

执行总和累计聚合得到的响应如下:

```

{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550},
          "cumulative_sales": {
            "value": 550
          }
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {"value": 60},
          "cumulative_sales": {
            // 这个值是1月加上2月的值
            "value": 610
          }
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": {"value": 375},
          "cumulative_sales": {"value": 985} // 这个值是上月求和的结果再加上3月的值
        }
      ]
    }
  }
}

```


} 在直方图聚合里随意加入普通的度量聚合，最后把移与平均数聚合放到直方图中。
 然后 buckets_path 参数“指出”一个直方图聚合中的兄弟度量聚合。

5.4.5 最大分组聚合

最大分组聚合所定义的分组包含一组聚合指定指标的最大值，并且同时输出分组的键和值。指定的指标必须是数字型而且这个组聚合必须是多组聚合。

用 max_bucket 来表示最大分组聚合，其结构如下：

```
{
  "max_bucket": {"buckets_path": "the_sum"}
}
```

参数见 5.4.1 节。

计算所有月份销售总额的最大值如下所示：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {"field" : "date", "interval" : "month"},
      "aggs": {
        "sales": {
          "sum": {"field": "price"}
        }
      }
    },
    "max_monthly_sales": {
      "max_bucket": {
        "buckets_path": "sales_per_month>sales"
      }
    }
  }
}
```

返回值如下：

```
{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550}
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
```

```

    "sales": {"value": 60}
  },
  {
    "key_as_string": "2015/03/01 00:00:00",
    "key": 1425168000000,
    "doc_count": 2,
    "sales": {"value": 375}
  }
]
},
"max_monthly_sales": { // 在上面的3个月中,最大的销售是1月的550
  "keys": ["2015/01/01 00:00:00"],
  "value": 550
}
}
}

```

5.4.6 最小分组聚合

最小分组聚合所定义的分组包含一组聚合指定指标的最小值,并且同时输出分组的键和值。指定的指标必须是数字型而且这个组聚合必须是多组聚合。

用 `min_bucket` 表示最小分组聚合,其结构如下:

```

{
  "min_bucket": {"buckets_path": "the_sum"}
}

```

参数见 5.4.2 节。

计算所有月份销售总额的最小值如下所示:

```

"aggs": {
  "sales_per_month": {
    "date_histogram": {"field": "date", "interval": "month"},
    "aggs": {
      "sales": {
        "sum": {"field": "price"}
      }
    }
  },
  "min_monthly_sales": {
    "min_bucket": {"buckets_path": "sales_per_month>sales"}
  }
}
}

```

返回值如下:

```

{
  "aggregations": {

```

```

    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550}
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {"value": 60}
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": {"value": 375}
        }
      ],
      "min_monthly_sales": { // 在上面的3个月中,最小的销售是2月的60
        "keys": ["2015/02/01 00:00:00"],
        "value": 60
      }
    }
  }
}

```

5.4.7 统计分组聚合

统计分组聚合,在一组聚合的所有分组中,对一个指定的指标计算各种统计值。指定的指标必须是数字型而且这个组聚合必须是多组聚合。

用 `stats_bucket` 来表示统计分组聚合,其结构如下:

```

{
  "stats_bucket": {"buckets_path": "the_sum"}
}

```

参数见 5.4.1 节。

计算所有月份销售总额的统计值如下所示:

```

{
  "aggs": {
    "sales_per_month": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      },
      "stats": {

```

```

    "sales": {
      "sum": {"field": "price"}
    },
    "stats_monthly_sales": {
      "stats_bucket": {"buckets_paths": "sales_per_month>sales"}
    }
  }
}

```

返回值如下：

```

{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {
            "value": 550
          }
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {
            "value": 60
          }
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": {
            "value": 375
          }
        }
      ]
    },
    "stats_monthly_sales": { // 统计的内容
      "count": 3,
      "min": 60,
      "max": 550,
      "avg": 328.333333333,
      "sum": 985
    }
  }
}

```

5.4.8 百分位分组聚合

百分位分组聚合是在一组聚合的所有分组中对一个指定的指标计算百分比。指定的指标必须是数字型而且这个组聚合必须是多组聚合。

用 `percentiles_bucket` 来表示百分比分组聚合，其结构如下：

```
{
  "percentiles_bucket": {
    "buckets_path": "the_sum"
  }
}
```

参数见 5.4.1 节，其中 `percents` 为进行计算的百分比列表（默认为 [1,5,25,50,75,95,99]）。对所有月份的销售总额进行百分比计算：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "sales": {
          "sum": {"field": "price"}
        }
      },
      "sum_monthly_sales": {
        "percentiles_bucket": {
          "buckets_paths": "sales_per_month>sales",
          "percents": [ 25.0, 50.0, 75.0 ]
        }
      }
    }
  }
}
```

返回值如下：

```
{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550}
        },
        {

```



```

    "key_as_string": "2015/02/01 00:00:00",
    "key": 1422748800000,
    "doc_count": 2,
    "sales": {"value": 60}
  },
  {
    "key_as_string": "2015/03/01 00:00:00",
    "key": 1425168000000,
    "doc_count": 2,
    "sales": {"value": 375}
  }
]
},
"percentiles_monthly_sales": {
  "values" : {"25.0": 60, "50.0": 375, "75.0": 550}
}
}
}

```

5.4.9 差值聚合

差值聚合 (derivative) 是计算一个指定指标两个分组之间的差值。指定的指标必须是数字型而且必须设置直方图 min_doc_count 参数为 0。

备注：英文 derivative 在此翻译成差值聚合更容易理解。

用 derivative 来表示差值聚合，其结构如下：

```

{
  "derivative": {
    "buckets_path": "the_sum"
  }
}

```

参数见 5.4.1 节。

1. 一级差值

计算所有月份销售总额的差值值如下所示：

```

{
  "aggs": {
    "sales_per_month": {
      "date_histogram": {
        "field": "date",
        "interval": "month"
      },
      "aggs": {
        "sales": {"sum": {"field": "price"}},
        "sales_deriv": {"derivative": {"buckets_path": "sales" }}
      }
    }
  }
}

```

5.4.3 百分位分组聚合

返回值如下:

```
{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550}
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {"value": 60},
          "sales_deriv": {"value": -490}
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": {"value": 375},
          "sales_deriv": {"value": 315}
        }
      ]
    }
  }
}
```

我们需要至少 2 个数据点来计算差值, 所以第一个分组没有差值。

差值的单位默认和 sales 聚合以及父直方图相同。所以在这种情况下, 如果价格字段的单位是美元, 差值的单位就是美元/月。

2. 二级差值

可以把差值管道聚合链接到另一个差值管道聚合的结果, 计算二级差值。示例如下:

```
{
  "aggs": {
    "sales_per_month": {
      "date_histogram": {"field": "date", "interval": "month"},
      "aggs": {
        "sales": {"sum": {"field": "price"}},
        "sales_deriv": {"derivative": {"buckets_path": "sales"}},
        "sales_2nd_deriv": {"derivative": {"buckets_path": "sales_deriv"}}
      }
    }
  }
}
```

```

    }
  }
}

```

二级差值聚合的参数 `buckets_path` 指向一级差值。

返回值如下：

```

{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": {"value": 550}
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": {"value": 60},
          "sales_deriv": {"value": -490}
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": {"value": 375},
          "sales_deriv": {"value": 315},
          "sales_2nd_deriv": {"value": 805}
        }
      ]
    }
  }
}

```

直到二级获取到 2 个以上的数据时，才会有二级差值。这意味着一级聚合产生了两个差值之后，才会产生二级聚合的值。

3. 单位

在差值聚合中，差值的单位是可以指定的。在响应中会返回一个额外的字段 `normalized_value`，汇报在 X 轴单位下的差值。

我们计算每个月的销售总额的差值，但是要求差值的单位是每天的销售总额，如下所示：

```

{
  "aggs" : {

```

```

    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "sales": { "sum": { "field": "price" } },
        "sales_deriv": {
          "derivative": { "buckets_path": "sales", "unit": "day" }
        }
      }
    }
  }
}

```

unit 指定用于差值计算的 X 轴的单位。

返回值如下：

```

{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "sales": { "value": 550 }
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2,
          "sales": { "value": 60 },
          "sales_deriv": {
            "value": -490,
            "normalized_value": -17.5
          }
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "sales": { "value": 375 },
          "sales_deriv": {
            "value": 315,
            "normalized_value": 10.16129032258065
          }
        }
      ]
    }
  }
}

```

```
}
}
```

value 值是原单位：每月。

normalized_value 值是请求的单位：每天。

5.4.10 分组脚本聚合

分组脚本聚合是父管道聚合，过程是执行一个脚本，使每个分组在父多组聚合的指定指标上执行计算。指定的指标必须是数字型而且脚本必须返回一个数字型的值。

用 bucket_script 来表示分组脚本聚合，其结构如下：

```
{
  "bucket_script": {
    "buckets_path": {
      "my_var1": "the_sum",
      "my_var2": "the_value_count"
    },
    "script": "my_var1 / my_var2"
  }
}
```

在这里，my_var1 是用于脚本中的分组路径变量名；the_sum 是用于变量的指标路径；Script 是聚合运行的脚本。脚本可以是内联、文件或者索引（必要参数）。其他参数见 5.4.1 节。

利用分组脚本聚合计算每个月 T 恤销售额占总量的百分比，如下所示：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "total_sales": { // 计算总数
          "sum": {"field": "price"}
        },
        "t-shirts": { t-shirts 总数
          "filter": {
            "term": {
              "type": "t-shirt"
            }
          },
          "aggs": {
            "sales": {
              "sum": {"field": "price"}
            }
          }
        }
      }
    }
  }
}
```



```

    }
  },
  "t-shirt-percentage": { // 用脚本计算百分比
    "bucket_script": {
      "buckets_path": {
        "tShirtSales": "t-shirts>sales",
        "totalSales": "total_sales"
      },
      "script": "tShirtSales / totalSales * 100"
    }
  }
}
}
}
}

```

返回值如下:

```

{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "total_sales": {"value": 50},
          "t-shirts": {
            "doc_count": 2,
            "sales": {"value": 10}
          },
          "t-shirt-percentage": {
            "value": 20
          }
        },
        {
          "key_as_string": "2015/02/01 00:00:00",
          "key": 1422748800000,
          "doc_count": 2
          "total_sales": {"value": 60},
          "t-shirts": {
            "doc_count": 1,
            "sales": {"value": 15}
          },
          "t-shirt-percentage": {"value": 25}
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "total_sales": {"value": 40},
          "t-shirts": {

```

```

    "doc_count": 1,
    "sales": {"value": 20}
  },
  "t-shirt-percentage": {"value": 50}
}
]
}
}

```

5.4.11 串行差分聚合

串行差分聚合是一个机制，用时间序列上的值减去不同时间区间或时间周期的值。例如，数据点 $f(x) = f(xt) - f(xt-n)$ ，其中， n 代表经过的时间。

时间段为 1 相当于没有时间规范的派生：只是把一个点换到下一个点。单时间周期可以用来移除常量，线性趋势。单周期也可以用来将数据转换为固定序列。

用 `serial_diff` 来表示串行差分聚合，其结构如下：

```

{
  "serial_diff": {"buckets_path": "the_sum", "lag": "7"}
}

```

参数见表 5-2。

表 5-2 串行差分聚合参数

参数名	描述	是否必要	默认值
<code>buckets_path</code>	指标路径	必要	
<code>lag</code>	当前分组覆盖历史分组。必须是非零的正整数	可选	1
<code>gap_policy</code>	决定遇到空白数据采取的行为	可选	<code>insert_zero</code>
<code>format</code>	聚合输出值的格式	可选	<code>null</code>

串行差分聚合必须嵌入到直方图或日期直方图聚合中使用，如下所示：

```

{
  "aggs": {
    "my_date_histo": {
      "date_histogram": {"field": "timestamp", "interval": "day"},
      "aggs": {
        "the_sum": {"sum": {"field": "lemmings"}},
        "thirtieth_difference": {
          "serial_diff": {"buckets_path": "the_sum", "lag": 30}
        }
      }
    }
  }
}

```

5.4.12 分组选择器聚合

分组选择器聚合是指在聚合的结果执行一个脚本来决定当前的分组是否应该保留。指定的指标必须是数字型而且脚本必须返回一个布尔值。如果脚本语言是表达式，一个数字型的返回值也是合法的。在这种情况下，0.0 将被视作假，其他的值被视作真。



注意 和所有的管道聚合一样，分组选择器聚合的结果过滤是在第一次聚合之后执行。这意味着利用分组选择器聚合过滤响应中返回的分组不会节省聚合执行时间。

用 `bucket_selector` 来表示分组选择器聚合，其结构如下：

```
{
  "bucket_selector": {
    "buckets_path": {
      "my_var1": "the_sum",
      "my_var2": "the_value_count"
    },
    "script": "my_var1 > my_var2"
  }
}
```

在这里，`my_var1` 是用在脚本中的分组路径变量名；`the_sum` 是用于变量的指标路径；Script 是聚合运行的脚本。脚本可以是内联、文件或者索引（必要参数）。其他参数见 5.4.1 节。仅保留每月销售总额小于等于 50 的分组，如下所示：

```
{
  "aggs" : {
    "sales_per_month" : {
      "date_histogram" : {
        "field" : "date",
        "interval" : "month"
      },
      "aggs": {
        "total_sales": {
          "sum": {"field": "price"}
        },
        "sales_bucket_filter": {
          "bucket_selector": {
            "buckets_path": {"totalSales": "total_sales"},
            "script": "totalSales <= 50"
          }
        }
      }
    }
  }
}
```

返回的响应如下:

```
{
  "aggregations": {
    "sales_per_month": {
      "buckets": [
        {
          "key_as_string": "2015/01/01 00:00:00",
          "key": 1420070400000,
          "doc_count": 3,
          "total_sales": {"value": 50}
        },
        {
          "key_as_string": "2015/03/01 00:00:00",
          "key": 1425168000000,
          "doc_count": 2,
          "total_sales": {"value": 40}
        }
      ]
    }
  }
}
```

5.5 小结

本章展示了 Elasticsearch 的强大数据分析能力: 聚合, 它可以对搜索查询的结果数据进行分组。利用简单的聚合, 建立复杂的数据分组。

分片选择器聚合的结果执行一个脚本来决定当前的分片是否应该保留。指定的指标必须是数字，而且必须返回一个布尔值。如果脚本返回一个数字类型的返回值也是合法的。在返回值为 0.0 的情况下，0.0 将被视为假，其他的值被视作真。

Chapter 6 第 6 章 集群管理

本章将介绍与集群相关的内容：集群节点监控、集群分片迁移、集群节点配置、节点发现、集群平衡位置等。Elasticsearch 正是利用集群进行水平扩展节点，达到支持海量数据的能力。

6.1 集群节点监控

在 Elasticsearch 的运行期间，一个很重要的方面就是监控。这使得系统管理员能够检测并预防可能性的问题，或至少知道失败时会发生什么。Elasticsearch 提供了非常详细的信息，使你能够检查和监控单个节点或一个整体的集群。包括集群的健康值、有关服务器的信息、节点信息、索引和分片信息等。对 Elasticsearch 监控的 API 主要有三类：一类是集群相关的，以 `_cluster` 开头，第二类是监控节点相关的，以 `_nodes` 开头，第三类是任务相关的，以 `_tasks` 开头。

6.1.1 集群健康值

集群健康值可以通过集群健康检查 API `_cluster/health` 得到简单情况。

请求：GET `http://127.0.0.1:9200/_cluster/health?pretty=true`

返回值：

```
{
  "cluster_name" : "elasticsearch",
  "status" : "yellow",
  "timed_out" : false,
  "number_of_nodes" : 1,
```



```

"number_of_data_nodes" : 1,
"active_primary_shards" : 16,
"active_shards" : 16,
"relocating_shards" : 0,
"initializing_shards" : 0,
"unassigned_shards" : 15,
"delayed_unassigned_shards" : 0,
"number_of_pending_tasks" : 0,
"number_of_in_flight_fetch" : 0,
"task_max_waiting_in_queue_millis" : 0,
"active_shards_percent_as_number" : 51.61290322580645
}

```

如果请求的后面加上索引的名称, 则可以得到这个索引的健康检查情况。例如:

请求: GET http://127.0.0.1:9200/_cluster/health/secisland?pretty=true

返回值中, status 字段提供的值反应了集群整体的健康程度, 它的状态是由系统中最差的分片决定的。值的意义如下:

- green——所有的主分片 (Primary Shard) 和副本分片 (Replica Shard) 都处于活动状态。
- yellow——所有的主分片都处于活动状态, 但是并不是所有的副本分片都处于活动状态。
- red——不是所有的主分片都处于活动状态。

6.1.2 集群状态

整个集群的综合状态信息是可以通过 _cluster/state 参数查询的。

请求: GET http://127.0.0.1:9200/_cluster/state

```

{
  "cluster_name": "elasticsearch",
  "version": 24,
  "state_uuid": "qgy8HW89RqmQPjhXAgYWjQ",
  "master_node": "100K_m-cSjecB6NbKJx9ew",
  "blocks": {},
  "nodes": {
    "100K_m-cSjecB6NbKJx9ew": {
      "name": "secilog",
      "transport_address": "127.0.0.1:9300",
      "attributes": {}
    }
  },
  "metadata": {...}, // 索引结构相关的信息
  "routing_table": {...}, // 索引分片相关的信息
  "routing_nodes": {...} // 路由节点相关的信息
}

```

默认情况下, 集群状态请求的是主节点的状态。在 state 后面增加 metadata 只请求索引结构相关的信息, 如果后面加上具体的索引, 则只请求这个具体索引结构相关的信息。

例如: 其他的几个例子:

❑ GET http://127.0.0.1:9200/_cluster/state/metadata, routing_table/foo, bar

❑ GET http://127.0.0.1:9200/_cluster/state/_all/foo, bar

❑ GET http://127.0.0.1:9200/_cluster/state/blocks

6.1.3 集群统计

集群统计 API `_cluster/stats` 可以从一个集群的角度来统计集群状态。它返回两个最基本的信息，一个是索引的信息，比如分片的数量、存储的大小、内存的使用等；另一个是集群节点的信息，比如节点数量、角色、操作系统信息、JVM 版本、内存使用率、CPU 和插件的安装信息。

请求：GET http://127.0.0.1:9200/_cluster/stats

返回值：

```
{
  "timestamp": 1475659140595,
  "cluster_name": "elasticsearch",
  "status": "yellow",
  "indices": { ... }, ①索引的统计信息
  "nodes": { ... } ②节点的统计信息
}
```

6.1.4 集群任务管理

集群任务管理接口是 2.3.0 新增接口，这个接口是实验性质的，未来有可能会改变。

请求如下：

```
GET /_tasks
GET /_tasks?nodes=nodeId1,nodeId2
GET /_tasks?nodes=nodeId1,nodeId2&actions=cluster:*
GET /_tasks/taskId1
GET /_tasks?parent_task_id=parentTaskId1
```

返回值：

```
{
  "nodes" : {
    "oTUltX4IQMOUUVeiohTt8A" : {
      "name" : "Tamara Rahn",
      "transport_address" : "127.0.0.1:9300",
      "host" : "127.0.0.1",
      "ip" : "127.0.0.1:9300",
      "tasks" : {
        "oTUltX4IQMOUUVeiohTt8A:124" : {
          "node" : "oTUltX4IQMOUUVeiohTt8A",
          "id" : 124,
          "type" : "direct",
          "action" : "cluster:monitor/tasks/lists[n]", // 具体的任务请求
        }
      }
    }
  }
}
```

```

    "start_time_in_millis" : 1458585884904,
    "running_time_in_nanos" : 47402,
    "parent_task_id" : "oTultX4IQMOUUVeiohTt8A:123"
  },
  "oTultX4IQMOUUVeiohTt8A:123" : {
    "node" : "oTultX4IQMOUUVeiohTt8A",
    "id" : 123,
    "type" : "transport",
    "action" : "cluster:monitor/tasks/lists", // 具体的任务请求
    "start_time_in_millis" : 1458585884904,
    "running_time_in_nanos" : 236042
  }
}
}
}
}

```

任务应用接口还可以用来等待任务的完成，下面的调用将等待 10 秒或 ID 为 otultx4-iqmouuveiohtt8a:12345 的任务等待 10 秒：

```
GET /_tasks/oTultX4IQMOUUVeiohTt8A:12345?wait_for_completion=true&timeout=10s
```

需要长时间等待的任务可以用下面的接口取消，取消操作可以取消一个或者多个任务：

```
POST /_tasks/taskId1/_cancel
```

```
POST /_tasks/_cancel?node_id=nodeId1,nodeId2&actions=*reindex
```

6.1.5 待定集群任务

待定集群任务接口返回一个尚未执行的集群水平变化的列表，例如，创建索引、更新映射、分配分片或失败分片。

请求：http://127.0.0.1:9200/_cluster/pending_tasks

它通常返回空，因为系统正常情况下处理是很快的。

6.1.6 节点信息

集群节点信息接口用于搜索一个或多个集群节点信息。节点信息的接口参数为 `_nodes`。

例如：

请求：[GET http://127.0.0.1:9200/_nodes](http://127.0.0.1:9200/_nodes)

返回值：

```

{
  "cluster_name": "elasticsearch",
  "nodes": {
    "I00K_m-cSjecB6NbKJx9ew": {
      "name": "secilog",
      "transport_address": "127.0.0.1:9300",
      "host": "127.0.0.1",

```

```

"ip": "127.0.0.1",
"version": "2.3.0",
"build": "8371be8",
"http_address": "127.0.0.1:9200",
"settings": {...},
"os": {...},
"process": {...},
"jvm": {...},
"thread_pool": {...},
"transport": {...},
"http": {...},
"plugins": [...],
"modules": [...]
}
}
}

```

从指标上来看，有 settings、os、process、jvm、thread_pool、transport、http 和 plugins。和集群统计类似，也可以在最后面加上指标信息，对关注的指标单独查看，比如 nodes/os 只查看操作系统信息。

活跃线程查看

通过 _nodes/hot_threads 接口查询活跃线程，在 _nodes 后面加上节点名可以查询某个节点上的活跃线程。

请求：GET http://127.0.0.1:9200/_nodes/hot_threads

或者请求具体某个节点：GET http://127.0.0.1:9200/_nodes/{nodesIds}/hot_threads。

6.1.7 节点统计

集群节点统计 API 可以对一个、多个或全部的集群节点进行统计。例如：

请求：GET http://127.0.0.1:9200/_nodes/stats/

返回值：

```

{
  "cluster_name": "elasticsearch",
  "nodes": {
    "I00K_m-cSjecB6NbKJx9ew": {
      "timestamp": 1475660864867,
      "name": "secilog",
      "transport_address": "127.0.0.1:9300",
      "host": "127.0.0.1",
      "ip": [
        "127.0.0.1:9300",
        "NONE"
      ],
      "indices": {...}, "os": {...},
      "process": {...}, "jvm": {...},

```

```

"thread_pool": {...}, "fs": {...},
"transport": {...}, "http": {...},
"breakers": {...}, "script": {...}
}
}

```

请求: GET http://127.0.0.1:9200/_nodes/Artemis/stats/

Artemis 是节点名称, 多个节点可以用逗号分开。这个命令把所有的统计信息都返回了, 可以在最后面加上指标, 只返回你关注指标的统计, 多个指标可以用逗号分开, 例如:

请求: GET http://127.0.0.1:9200/_nodes/Artemis/stats/os,fs

上面的命令只返回节点中操作系统和文件系统的信息, 统计指标解释如下:

- ☐ indices——索引相关的数据统计, 包括 size、document count、indexing and deletion times、search times、field cache size、merges and flushes。
- ☐ fs——文件系统信息, 包括路径、磁盘空间、读 / 写数据等。
- ☐ http——HTTP 连接信息。
- ☐ jvm——JVM 虚拟机的统计, 包括内存池的数据、垃圾回收、缓冲池一些加载 / 卸载类等。
- ☐ os——操作系统统计, 平均负载、内存、交换分区等。
- ☐ process——进程统计, 包括内存消耗、处理器使用、打开文件描述符等。
- ☐ thread_pool——线程池的统计, 包括当前大小、队列和拒绝的任务等。
- ☐ transport——网络数据统计, 集群通信中发送和接收字节的统计。
- ☐ breaker——数据处理的统计。
- ☐ script——脚本相关的信息。

6.2 集群分片迁移

在 Elasticsearch 中可以通过集群路由 API `_cluster/reroute` 来对集群中的分片进行操作, 例如可以在集群中把一个分片从一个节点迁移到另一个节点, 将未分配的分片可以分配到一个特定节点上等。不过要想完全手动, 必须先把 `cluster.routing.allocation.disable_allocation` 参数设置为 `true`, 禁止 ES 进行自动索引分片分配, 否则你从 A 节点把分片移到 B 节点, 那么另外一个节点的一个分片又会移到 A 节点。语法如下:

请求: POST http://127.0.0.1:9200/_cluster/reroute

```

{
  "commands": [ {
    "move": {
      "index": "test", "shard": 0,
      "from_node": "node1", "to_node": "node2"
    }
  },

```



```
{
  "allocate" : { "index" : "test", "shard" : 1, "node" : "node3" }
}
]
```

一共有三种操作移动 (move)、取消 (cancel) 和分配 (allocate)。下面分别介绍这三种情况:

- ❑ 移动 (move) —— 把分片从一个节点移动到另一个节点。可以指定索引名和分片号。
- ❑ 取消 (cancel) —— 取消分配一个分片。可以指定索引名和分片号。node 参数可以指定在哪个节点取消正在分配的分片。allow_primary 参数支持取消分配主分片。
- ❑ 分配 (allocate) —— 分配一个未分配的分片到指定节点, 可以指定索引名和分片号。node 参数指定分配到哪个节点; allow_primary 参数可以强制分配主分片, 不过这样可能导致数据丢失。因为 allow_primary 参数将使一个新的空主分片被分配, 如果有一份原始的分片副本 (包括数据) 的节点重新加入集群, 分片副本数据将被删除: 旧的分片副本将会被新的活动分片副本替换。

集群配置更新: 系统允许对集群的配置进行更新。配置更新有两种状态, 一是持久的, 二是临时的, 例如:

请求: PUT 127.0.0.1:9200/_cluster/settings

```
{
  "persistent" : { "discovery.zen.minimum_master_nodes" : 2 }
}
```

上面的例子是持久更新配置, 如果把参数 persistent 换成了 transient, 就是临时更新。

6.3 集群节点配置

当我们启动 Elasticsearch 的实例, 就会启动至少一个节点。相同集群名的多个节点的连接就组成了一个集群, 在默认情况下, 集群中的每个节点都可以处理 HTTP 请求和集群节点的数据传输。集群中所有的节点都知道集群中其他所有的节点, 可以将客户端请求转发到适当的节点。节点有以下类型:

- ❑ **主 (master) 节点:** 在一个节点上 node.master 设置为 true (默认) 的时候, 它有资格被选作为主节点, 控制整个集群。
- ❑ **数据 (data) 节点:** 在一个节点上 node.data 设置为 true (默认) 的时候。该节点保存数据和执行数据相关的操作, 如增删改查、搜索和聚合。

默认情况下, 节点同时是主节点和数据节点, 这是非常方便的小集群, 但随着集群的发展, 分离主节点和数据节点将变得非常重要。

- ❑ **客户端节点:** 当一个节点的 node.master 和 node.data 都设置为 false 的时候, 它既不

能保持数据也不能成为主节点，该节点可以作为客户端节点，可以响应用户的请求，并把相关操作发送到其他节点。

□ 部落节点：当一个节点配置 `tribe.*` 的时候，它是一个特殊的客户端，它可以连接多个集群，在所有连接的集群上执行搜索和其他操作。

客户端节点在搜索请求或批量增加索引请求等可能涉及在不同数据节点上的操作。这些请求会分成两个阶段，一是接收客户端的请求，二是协调节点执行相关操作。当数据分散在不同的节点上时，协调节点将请求转发到数据节点，每个数据节点在本地执行请求并把结果传输给协调节点，然后协调节点收集各个数据节点的结果转换成单个请求结果返回。所以需要客户端有足够的内存和 CPU 来处理各个节点的返回结果。

6.3.1 主节点

主节点的主要职责是和集群操作相关的内容，如创建或删除索引，跟踪哪些节点是集群的一部分，并决定哪些分片分配给相关的节点。稳定的主节点对集群的健康是非常重要的。默认情况下任何一个集群中的节点都有可能被选为主节点。索引数据和搜索查询等操作会占用大量的 CPU、内存、IO 资源，为了确保一个集群的稳定，分离主节点和数据节点是一个比较好的选择。

虽然主节点也可以协调节点，搜索路由和从客户端新增数据到数据节点，但最好不要使用这些专用的主节点。一个重要的原则是，尽可能做少量的工作。创建一个独立的主节点的配置如下：

```
node.master: true
node.data: false
```

为了防止数据丢失，配置 `discovery.zen.minimum_master_nodes` 是至关重要的（默认为 1），每个主节点应该知道形成一个集群的最小数量的主资格节点的数量。解释一下。

假设我们有一个集群，有 3 个主节点，当网络发生故障的时候，有可能其中一个节点不能和其他节点进行通信了。这个时候，当 `discovery.zen.minimum_master_nodes` 设置为 1 时，就会分成两个小的独立集群，当网络好的时候，就会出现数据错误或者丢失数据的情况。当 `discovery.zen.minimum_master_nodes` 设置为 2 的时候，一个网络中有两个主节点，可以继续工作，另一部分，由于只有一个主资格节点，则不会形成一个独立的集群，这个时候当网络回复的时候，节点又会重新加入集群。设置这个值的原则是：

$$(\text{master_eligible_nodes} / 2) + 1.$$

这个参数也可以动态设置：

请求：PUT `http://127.0.0.1:9200/_cluster/settings`

```
{
  "transient": {"discovery.zen.minimum_master_nodes": 2}
}
```

6.3.2 数据节点

数据节点主要是存储索引数据的节点，主要对文档进行增删改查、聚合操作等。数据节点对 CPU、内存、IO 要求较高，在优化的时候需要监控数据节点的状态，当资源不够的时候，需要在集群中添加新的节点。数据节点的配置如下：

```
node.master: false
node.data: true
```

6.3.3 客户端节点

当主节点和数据节点配置都设置为 false 时，该节点只能处理路由请求，处理搜索，分发索引操作等，从本质上来说该客户端节点表现为智能负载均衡器。独立的客户端节点在一个比较大的集群中是非常有用的，它协调主节点和数据节点，客户端节点加入集群可以得到集群的状态，根据集群的状态可以直接发送路由请求。



警告

添加太多的客户端节点对集群是一种负担，因为主节点必须等待每一个节点集群状态的更新确认！客户端节点的作用不应被夸大，数据节点也可以起到类似的作用。

配置如下：

```
node.master: false
node.data: false
```

数据节点路径设置：每一个主节点和数据节点都需要知道分片、索引、元数据的物理存储位置，`path.data` 默认位为 `$ES_HOME/data`，可以通过配置文件 `elasticsearch.yml` 进行修改，例如：

```
path.data: /var/elasticsearch/data
```

这个设置也可以在命令行上执行，例如：

```
elasticsearch --path.data /var/elasticsearch/data
```

这个路径最好进行单独配置，这样 Elasticsearch 的目录和数据的目录就会分开。当删除了 Elasticsearch 主目录的时候，不会影响到数据。通过 rpm 安装默认是分开的。

数据目录可以被多个节点共享，甚至可以属于不同的集群，为了防止多个节点共享相同的数据路径，可以在配置文件 `elasticsearch.yml` 中添加：`node.max_local_storage_nodes: 1`



注意

在相同的数据目录不要运行不同类型的节点（例如：`master`、`data`、`client`），这会导致意外的数据丢失。

6.3.4 部落节点

部落节点可以跨越多个集群，它可以接收每个集群的状态，然后合并成一个全局集群的状态，它可以读写所有节点上的数据，部落节点在 `elasticsearch.yml` 中的配置如下：

```
tribe:
  t1:
    cluster.name: cluster_one
  t2:
    cluster.name: cluster_two
```

T1 和 T2 是任意的名字，代表连接到每个集群。上面的示例配置两个集群连接，名称分别是 T1 和 T2。默认情况下，部落节点通过单播发现来连接每一个集群。用于连接的任何其他设置可以配置在 `tribe{name}` 后，就像上面的例子。大多数情况下，部落节点可以像单节点一样对集群进行操作。

注意，以下操作将和单节点操作不同，如果两个集群的名称相同，部落节点只会连接其中一个。

由于没有主节点，主节点级别的读操作如：集群统计、集群健康度，会在本地标志设置为 `true` 的节点自动执行。

主节点级别的写操作将被拒绝，如创建索引，这些操作应该在单个集群节点上执行。

部落节点可以通过块 (block) 设置所有的写操作和所有的元数据操作，例如：

```
tribe:
  blocks:
    write: true
    metadata: true
```

部落节点也可以在选中的索引块中单独配置以上操作，例如：

```
tribe:
  blocks:
    write.indices: hk*,ldn*
    metadata.indices: hk*,ldn*
```

当多个集群有相同的索引名的时候，默认情况下，部落节点将选择其中一个。这可以通过 `tribe.on_conflict setting` 进行配置，可以设置排除那些索引或者指定固定的部落名称。

6.4 节点发现

在 Elasticsearch 中，节点之间可以相互发现，并把相同集群名称的节点统一成一个集群，那节点是如何发现的呢，这次就谈一下节点发现的一些内部细节。在 Elasticsearch 内部，zen 发现机制是默认的发现模块。它提供了单播发现方式，能够很容易地扩展至云环境。发现模块和其他模块集成，例如所有节点的通信是通过传输模块。发现模块分成两个模块：

□ ping 模块，通过 ping 模块可以寻找其他节点。

❑ 单播模块，需要提供一个主机列表作为路由列表。以 `discovery.zen.ping.unicast` 作为前缀。

主机列表设置：`hosts`，配置全路径为：`discovery.zen.ping.unicast.hosts`。它是一个数组的配置，多个主机设置以逗号分开。格式为 `host:port`，或者 `host[port1-port2]`，ip6 的主机必须放在方括号内，例如 `127.0.0.1, [::1]`。

单播发现依赖传输模块实现。注意 `port` 默认端口是 9300。

下面介绍主节点选举过程。

6.4.1 主节点选举

在集群中，系统会自动通过 `ping` 来进行选举主节点或者加入主节点，这些都是自动完成的：

❑ `discovery.zen.ping_timeout`（默认 3 秒）配置允许对选举的时间进行调整，用来处理缓慢或拥挤的网络。当一个节点请求加入主节点，它会发送请求信息到主节点，请求的超时时间配置为 `discovery.zen.join_timeout`，这个时间比较长，是 `discovery.zen.ping_timeout` 时间的 20 倍。

当主节点发生问题的时候，现有的节点又会通过 `ping` 来重新选举一个新的主节点。当 `discovery.zen.master_election.filter_client` 设置为 `true` 的时候，在选举主节点的时候从客户端节点 (`node.client` 为 `true`，或者 `node.data` 和 `node.master` 同时为 `false`) 的 `ping` 操作将被忽略，该参数默认为 `true`。当 `discovery.zen.master_election.filter_data` 为 `true` 时，在选举主节点的时候从数据节点 (`node.data` 为 `true`，`node.master` 同时为 `false`) 的 `ping` 操作将被忽略，默认为 `false`。主节点配置为 `true` 的节点一直都有选举的资格。

当节点 `node.master` 设置为 `false` 或者 `node.client` 设置为 `true` 的时候，它们将自动排除成为主节点的可能性。

❑ `discovery.zen.minimum_master_nodes` 设置需要加入一个新当选的主节点的最小节点数目，或者接收它作为主节点的最小节点数。如果不满足这一要求，主节点会下台，重新开始新的选举。

6.4.2 故障检测

有两种方式进行故障检测，第一个是由主节点到所有其他节点的验证，证明它们还活着。另一种，是每个节点 `ping` 主节点验证，当主节点有故障的时候会启动选举过程。控制故障检测过程使用 `discovery.zen.fd` 前缀设置：

❑ `ping_interval`：`ping` 检查的频率（时间间隔），默认值为 1 秒。

❑ `ping_timeout`：`ping` 的超时时间，默认为 30 秒。

❑ `ping_retries`：多少次 `ping` 失败或者超时的节点被认为是失败的。默认 3 次。

主节点是在一个集群中可以改变集群状态的唯一节点。主节点处理一个集群状态的更

新,适用于所需的更改,并将更新的集群状态发布到集群中的所有其他节点上。每个节点接收发布消息,更新它自己的集群状态,并对主节点进行应答,该主节点等待所有节点响应,然后在处理下一个更新的队列前进行超时处理。`discovery.zen.publish_timeout` 默认为 30 秒,可以通过动态配置进行设置。

一个健康的集群必须有一个主节点,并且有多个满足主节点选举条件的节点,这个数字由 `discovery.zen.minimum_master_nodes` 设置。`discovery.zen.no_master_block` 设置没有有效的主节点时应拒绝的操作,有两个选择:

- `all`: 这个节点上的所有行为被拒绝,包括读写,同时拒绝集群状态的读写操作,比如得到索引设置。
- `write`: 基于最后一次的集群配置,写操作被拒绝,允许读操作。这可能会导致部分读取的数据是过期的,因为这个节点已经从集群中分离,这个是默认设置。

`discovery.zen.no_master_block` 设置不适用于基于节点状态的 API 例如 `cluster stats`、`node info` 和 `node stats`。

6.5 集群平衡配置

一个主节点的主要作用是决定哪些分片分配给哪个节点,并在节点之间移动分片以平衡整个集群。有很多设置可以控制分片的分配过程:

- 分片分配设置——可以通过设置来对集群进行分配和再平衡操作。
- 基于磁盘的配置——可以根据可用磁盘空间大小来进行分配设置。
- 分片智能分配——可以控制分片在不同的机架或可用性区域中的配置。
- 分片配置过滤——允许某些节点或节点组排除在分配的列表中,常用在需要停止的节点中。

在本节中的所有设置都是动态设置,可以通过集群接口进行实时更新。

6.5.1 分片分配设置

分片分配过程是分片到节点的一个处理过程,它可能发生在初始恢复过程中,副本分配中,再平衡过程中,或当节点被添加或删除时。

1. 分片分配设置

下面的动态设置可以用来控制分片的分配和回收。

□ `cluster.routing.allocation.enable`: 禁用或启用哪种类型的分片,可选的参数有:

- `all`——允许所有的分片被重新分配。
- `primaries`——只允许主结点分片被重新分配。
- `new_primaries`——只允许新的主结点索引的分片被重新分配。

- none——不对任何分片进行重新分配。

- ❑ `cluster.routing.allocation.node_concurrent_recoveries` : 允许在一个节点上同时并发多少个分片分配, 默认为 2。
- ❑ `cluster.routing.allocation.node_initial primaries_recoveries` : 当副本分片加入集群的时候, 在一个节点上并行发生分片分配的数量, 默认是 4 个。
- ❑ `cluster.routing.allocation.same_shard.host` : 在一个主机上的当有多个相同的集群名称的分片分配时, 是否进行检查, 检查主机名和主机 ip 地址。默认为 false, 此设置仅适用于在同一台机器上启动多个节点时配置。
- ❑ `indices.recovery.concurrent_streams` : 从一个节点恢复的时候, 同时打开的网络流量的数量, 默认为 3。
- ❑ `indices.recovery.concurrent_small_file_streams` : 从同伴的分片恢复时打开每个节点的小文件 (小于 5M) 流的数目, 默认为 2。

2. 分片平衡设置

下面的动态设置可以用来控制整个集群的碎片再平衡, 配置有:

- ❑ `cluster.routing.rebalance.enable`: 启用或禁用特定种类的分片重新平衡, 可选的参数有:
 - all——允许所有的分片进行分片平衡, 默认配置。
 - primaries——只允许主分片进行平衡。
 - replicas——只允许从分片进行平衡。
 - none——不允许任何分片进行平衡。
- ❑ `cluster.routing.allocation.allow_rebalance`: 当分片再平衡时允许的操作, 可选的参数有:
 - always——总是允许再平衡。
 - `indices primaries_active`——只有主节点索引允许再平衡。
 - `indices_all_active`——所有的分片允许再平衡, 默认参数。
 - `cluster.routing.allocation.cluster_concurrent_rebalance` : 重新平衡时允许多少个并发的分片同时操作, 默认为 2。

3. 启发式分片平衡

以下设置用于确定在何处放置每个碎片的数据:

- ❑ `cluster.routing.allocation.balance.shard`: 在节点上分配每个分片的权重, 默认是 0.45。
- ❑ `cluster.routing.allocation.balance.index` : 在特定节点上, 每个索引分配的分片的数量, 默认 0.55。

`cluster.routing.allocation.balance.threshold`: 操作的最小最优化的值。默认为 1。

6.5.2 基于磁盘的配置

Elasticsearch 可以根据磁盘的大小来决定是否重新进行分片的分配。有如下配置:

□ `cluster.routing.allocation.disk.threshold_enabled` : 是否启用磁盘分配决策, 默认为 `true`。

□ `cluster.routing.allocation.disk.watermark.low` : 允许分配时的磁盘空间最小值, 可以是比例或者绝对值, 比如 85% 或者 1G。当磁盘占用超过设定的值之后, 系统将不会对此节点进行分配操作。

□ `cluster.routing.allocation.disk.watermark.high` : 允许保存分片节点磁盘空间的最大值, 当超过这个值后, 系统会把分片迁移到别的节点。默认 90%。也可以设置一个具体的大小值 (比如 500mb) 当可用空间小于这个值的时候, 系统会自动迁移分片到别的节点。

□ `cluster.info.update.interval` : 检查集群中每个节点的磁盘使用情况的时间间隔, 默认 30 秒。

□ `cluster.routing.allocation.disk.include_relocations` : 当计算节点的磁盘使用时需要考虑当前被分片的情况。默认为 `true`。

比如下面的一个配置实例:

请求: `PUT http://127.0.0.1:9200/_cluster/settings{`

```
"transient": {
  "cluster.routing.allocation.disk.watermark.low": "80%",
  "cluster.routing.allocation.disk.watermark.high": "5gb",
  "cluster.info.update.interval": "1m"
}
```

实例的含义是, 每分钟检查一下磁盘空间, 当磁盘已用空间小于 80% 的时候参与分片分配, 当可用空间不足 5GB 的时候, 将节点的分片迁移到别的节点。

需要注意的是, 在 2.0.0 版本前, 当系统有多个数据盘的时候, 系统考虑的是总大小, 在 2.0.0 版本之后, 系统考虑的是每个磁盘的使用情况。

6.5.3 分片智能分配

实际部署时, 很多时候是部署在虚拟机中, 共享同一个物理节点, 或者部署的时候在同一个机架、同一个网络区域中。当这些情况下遇到故障时, 很多节点会同时发生故障, 导致系统出现问题。如果在配置 Elasticsearch 的时候事先能够注意到把不同的节点分布在不同的物理机器、不同的机架或者不同的网络区域中, 这样当一个节点出现问题的时候, 会使风险降到最低。

区域分片分配 (Shard Allocation Awareness) 设置用于配置 Elasticsearch 关于硬件的信息, 例如, 我们在启动的时候在启动程序后面加上 `--node.rack_id`, 后面跟上一个指定的名称, 这个配置也可以放在配置文件中:

```
elasticsearch --node.rack_id rack_one
```

同时我们需要设置 `cluster.routing.allocation.awareness.attributes: rack_id`，可以在配置文件中设置，或者通过 `cluster-update-settings` API 接口设置。

假设有两个节点 `node.rack_id` 的名称都为 `rack_one`，我们创建一个索引有 5 个主要的分片和一个副本分片。将所有的主分片和副本分片分配在两个节点上。然后我们新加两个节点，节点 `node.rack_id` 的名称为 `rack_two`。这样配置后，系统会自动分配主分片到新的节点上，确保没有两个相同的分片在同一个区域中。

当搜索的时候，系统会智能地处理，只在一个区域中搜索，这样会比在不同的区域中搜索更快。

当使用区域分片分配属性，碎片不会分配给没有设置这些属性的值的节点。具有相同区域属性值的一组节点中主 / 副本分片的数量由属性值的数量决定。当一组节点的数量是不平衡的，并且有许多的副本，可能会产生未赋值的副本分片。

强制分配属性，解决了不允许相同的分片副本被分配到同一区域的问题。假设当我们有两个区域的时候，每个区域的大小只够分配一半的分片，如果一个区域不可用，全部分片都在一个区域会导致空间不够，引起系统异常，这个时候，强制分配属性就有意义了。示例配置如下：

```
cluster.routing.allocation.awareness.force.zone.values: zone1,zone2
cluster.routing.allocation.awareness.attributes: zone
```

这个时候，如果我们启动了 `zone1` 上的两个节点，并创建有 5 个主分片一个副本分片的索引，这个时候在 `zone1` 上只会启动主分片，只到 `zone2` 的节点启动后，才会启动副本分片。

6.5.4 分片配置过滤

分片配置过滤可以对索引的分片分配进行控制，允许或禁止分配到指定节点。这个功能用在集群级别分片分配过滤上，可在移除节点之前把节点上的分片移动到其他节点中。例如我们想停止 `10.0.0.1` 这个 IP 上的所有节点，可如下操作：

请求：PUT `http://127.0.0.1:9200/_cluster/settings`

```
{
  "transient" : {"cluster.routing.allocation.exclude_ip" : "10.0.0.1"}
}
```

这样正常情况下，`10.0.0.1` 节点上的分片会被迁移到其他节点。动态配置的属性如下：

- ❑ `cluster.routing.allocation.include.{attribute}`：将索引分配给一个节点，其 `{attribute}` 至少有一个逗号分隔的值。
- ❑ `cluster.routing.allocation.require.{attribute}`：将索引分配给一个节点，该节点的 `{attribute}` 具有所有的逗号分隔值。
- ❑ `cluster.routing.allocation.exclude.{attribute}`：将索引分配给一个节点，其 `{attribute}`

没有一个逗号分隔的值。

attributes 可以包含的值有：

- `_name`——通过节点名称匹配节点。
- `_ip`——通过 IP 地址匹配节点。
- `_host`——通过机器名称匹配节点。

所有的属性值可以用通配符，如：

请求：PUT `http://127.0.0.1:9200/_cluster/settings`

```
{
  "transient": {"cluster.routing.allocation.include_ip": "192.168.2.*"}
}
```

6.5.5 其他集群配置

还有一些集群配置如下。

只读设置：可以设置 `cluster.blocks.read_only` 使整个集群为只读。

日志记录：设置日志记录的级别，例如增加了 `indices.recovery` 模块，其日志记录级别的调试代码如下：

请求：PUT `http://127.0.0.1:9200/_cluster/settings`

```
{
  "transient": {"logger.indices.recovery": "DEBUG"}
}
```

6.6 小结

本章介绍了 Elasticsearch 中的集群接口，通过接口可以监控和配置集群。本章同时介绍了集群分配的原理，通过这些原理可以根据我们的实际情况设置集群节点的性质，设置集群的参数，为长期稳定运行 Elasticsearch 打下基础。

索引分词器

本章重点介绍 Elasticsearch 中的分词器，特别是中文分词器。通过本章介绍，可以了解 Elasticsearch 分词的原理、过程，以及如何添加新的分词器等，同时本章还介绍插件相关知识，及其功能与安装等。

7.1 分词器的概念

在 Elasticsearch 中，索引分析模块是可以通过注册分词器 (analyzer) 来进行配置。分词器的作用是当一个文档被索引的时候，分词器从文档中提取出若干词元 (token) 来支持索引的存储和搜索。

分词器 (analyzer) 是由一个分解器 (tokenizer)、零个或多个词元过滤器 (token filters) 组成。

分解器处理前可能要做一些预处理，比如去掉里面的 HTML 标记，这些处理的算法称为字符过滤器 (character filter)，一个分解器会有一个或多个字符过滤器。分解器是用来把字符串分解成一系列词元。一个简单的分解器是把一个句子当遇到空格或标点符号时，分解成一个个的索引词。Elasticsearch 内置了分词器、分解器和词元过滤器，同时也支持自定义这些内容。

词元过滤器的作用是对分词器提取出来的词元做进一步处理，比如转成小写，增加同义词等。处理后的结果称为索引词 (Term)，文档中包含了几个这样的 Term 称为 Frequency (词频)。引擎会建立 Term 和原文档的倒排索引 (Inverted Index)，这样就能根据 Term 很快找到源文档了。

Elasticsearch 内置了很多分词器，还有很多第三方的分词器插件，比如一些处理中文的

分词器（后面称为“中文分词器”）。

索引分析模块包括分词器（analyzer），分解器（tokenizer）和词元过滤器（token filters）。如果没有明确定义分析模块，系统会用内置的分词器、分解器和过滤器。下面是一个分词器的配置：

```
index :
  analysis : // 一个索引分析模块
    analyzer : // 一个分词器
      standard :
        type : standard
        stopwords : [stop1, stop2]
      myAnalyzer1 :
        type : standard
        stopwords : [stop1, stop2, stop3]
        max_token_length : 500
      # configure a custom analyzer which is
      # exactly like the default standard analyzer
      myAnalyzer2 :
        tokenizer : standard
        filter : [standard, lowercase, stop]
      tokenizer : // 一个分解器
        myTokenizer1 :
          type : standard
          max_token_length : 900
        myTokenizer2 :
          type : keyword
          buffer_size : 512
        filter : // 一个词元过滤器
          myTokenFilter1 :
            type : stop
            stopwords : [stop1, stop2, stop3, stop4]
          myTokenFilter2 :
            type : length
            min : 0
            max : 2000
```

系统有一个配置选项来定义哪个分词器将在默认情况下使用。默认的逻辑名称下有一个的分词器，将用于索引和搜索。配置参数为 `default_search`。例如：

```
index :
  analysis :
    analyzer :
      default :
        tokenizer : keyword
```

分词器可以通过别名来查找相关的分词器进行分词。例如，以下将允许标准分词器进行分词的同时也可参照别名为 `alias1` 和 `alias2` 的分词器进行分词。

```
index :
  analysis :
```

```

analyzer :
  standard :
    alias: [alias1, alias2]
    type : standard
    stopwords : [test1, test2, test3]

```

系统默认的分词器有: Standard Analyzer, Simple Analyzer, Whitespace Analyzer, Stop Analyzer, Keyword Analyzer, Pattern Analyzer, Language Analyzers, Snowball Analyzer, Custom Analyzer。

系统默认的分解器有: Standard Tokenizer, Edge NGram Tokenizer, Keyword Tokenizer, Letter Tokenizer, Lowercase Tokenizer, NGram Tokenizer, Whitespace Tokenizer, Pattern Tokenizer, UAX Email URL Tokenizer, Path Hierarchy Tokenizer, Classic Tokenizer, Thai Tokenizer。

系统默认的词元过滤器有: Standard Token Filter, ASCII Folding Token Filter, Length Token Filter, Lowercase Token Filter, Uppercase Token Filter, NGram Token Filter, Edge NGram Token Filter, Porter Stem Token Filter, Shingle Token Filter, Stop Token Filter, Word Delimiter Token Filter, Stemmer Token Filter, Stemmer Override Token Filter, Keyword Marker Token Filter, Keyword Repeat Token Filter, KStem Token Filter, Snowball Token Filter, Phonetic Token Filter, Synonym Token Filter, Compound Word Token Filter, Reverse Token Filter, Elision Token Filter, Truncate Token Filter, Unique Token Filter, Pattern Capture Token Filter, Pattern Replace Token Filter, Trim Token Filter, Limit Token Count Token Filter, Hunspell Token Filter, Common Grams Token Filter, Normalization Token Filter, CJK Width Token Filter, CJK Bigram Token Filter, Delimited Payload Token Filter, Keep Words Token Filter, Keep Types Token Filter, Classic Token Filter, Apostrophe Token Filter, Decimal Digit Token Filter。

系统默认的角色过滤器有: Mapping Char Filter, HTML Strip Char Filter, Pattern Replace Char Filter。

分词模块的使用还可以通过单独的索引分析接口来体现。例子如下。

请求: POST http://127.0.0.1:9200/_analyzer/

参数:

```

{
  "tokenizer": "standard",
  "filters": [
    "lowercase",
    "stop"
  ],
  "text": "SecIsland is a data company."
}

```

注意 "analyzer", "tokenizer", "filters" 三个是关键字。analyzer 在内容上包括 tokenizer

和 filters, 所以可以只输入 analyzer, 或者输入 tokenizer 和 filters。

返回值为:

```
{
  "tokens": [
    { "token": "secisland", "start_offset": 0, "end_offset": 9,
      "type": "<ALPHANUM>", "position": 0 },
    { "token": "data", "start_offset": 15, "end_offset": 19,
      "type": "<ALPHANUM>", "position": 3 },
    { "token": "company", "start_offset": 20, "end_offset": 27,
      "type": "<ALPHANUM>", "position": 4 }
  ]
}
```

自定义分词器

自定义分词器接受分解器的逻辑/注册名, 以及词元过滤器的逻辑/注册名列表。自定义分词器的名称不能以“_”开头。

自定义分词器类型可以有如下设置:

- tokenizer——分解器的逻辑/注册名。
- filter ——词元过滤器的逻辑/注册名的可选列表。
- char_filter——字符过滤器的逻辑/注册名的可选列表。
- position_increment_gap——每个使用本分词器字段的字段值之间可增加的位置, 默认为 100。

举个例子。

```
index :
  analysis :
    analyzer :
      myAnalyzer2 :
        type : custom // 自定义分词器
        tokenizer : myTokenizer1 // 分解器别名, 具体在下面定义
        filter : [myTokenFilter1, myTokenFilter2] // 词元过滤器别名
        char_filter : [my_html] // 字符过滤器别名
        position_increment_gap : 256
      tokenizer : // 具体定义分解器
        myTokenizer1 :
          type : standard
          max_token_length : 900
        filter : // 具体定义词元过滤器
          myTokenFilter1 :
            type : stop
            stopwords : [stop1, stop2, stop3, stop4]
          myTokenFilter2 :
            type : length
            min : 0
            max : 2000
```

```
char_filter : // 具体定义字符过滤器
my_html :
  type : html_strip
  escaped_tags : [xxx, yyy]
  read_ahead : 1024
```

7.2 中文分词器

在Elasticsearch中，内置了很多分词器(analyzers)，但默认的分词器对中文的支持都不是太好。所以需要单独安装插件来支持，比较常用的是中科院 ICTCLAS 的 smartcn 和 IKAnalyzer，效果还是不错的，但是目前 IKAnalyzer 还不支持最新的 Elasticsearch 2.3.0 版本，smartcn 中文分词器默认官方支持，它提供了一个中文或混合中文英文文本的分析器。支持最新的 2.3.0 版本。但是 smartcn 不支持自定义词库，作为测试可先用一下。后面介绍如何支持最新的版本。下面分别介绍两个中文分词器。

1. smartcn

安装分词：plugin install analysis-smartcn

卸载：plugin remove analysis-smartcn

测试：

请求：POST http://127.0.0.1:9200/_analyze/

```
{
  "analyzer": "smartcn",
  "text": "联想是全球最大的笔记本厂商"
}
```

返回值：

```
{
  "tokens": [
    {"token": "联想", "start_offset": 0, "end_offset": 2, "type": "word", "position": 0},
    {"token": "是", "start_offset": 2, "end_offset": 3, "type": "word", "position": 1},
    {"token": "全球", "start_offset": 3, "end_offset": 5, "type": "word", "position": 2},
    {"token": "最", "start_offset": 5, "end_offset": 6, "type": "word", "position": 3},
    {"token": "大", "start_offset": 6, "end_offset": 7, "type": "word", "position": 4},
    {"token": "的", "start_offset": 7, "end_offset": 8, "type": "word", "position": 5},
    {"token": "笔记本", "start_offset": 8, "end_offset": 11, "type": "word", "position": 6},
    {"token": "厂商", "start_offset": 11, "end_offset": 13, "type": "word", "position": 7}
  ]
}
```

作为对比，我们看一下标准分词的结果，在请求中把 smartcn，换成 standard 然后看返回值：

```
{
  "analyzer": "tokenizer",
  "filters": [
    "lowercase"
  ],
  "text": "联想是全球最大的笔记本厂商"
}
```



```

"tokens": [
  {"token": " 联 ", "start_offset": 0, "end_offset": 1, "type": "<IDEOGRAPHIC>", "position": 0},
  {"token": "想", "start_offset": 1, "end_offset": 2, "type": "<IDEOGRAPHIC>", "position": 1},
  {"token": "是", "start_offset": 2, "end_offset": 3, "type": "<IDEOGRAPHIC>", "position": 2},
  {"token": "全", "start_offset": 3, "end_offset": 4, "type": "<IDEOGRAPHIC>", "position": 3},
  {"token": "球", "start_offset": 4, "end_offset": 5, "type": "<IDEOGRAPHIC>", "position": 4},
  {"token": "最", "start_offset": 5, "end_offset": 6, "type": "<IDEOGRAPHIC>", "position": 5},
  {"token": "大", "start_offset": 6, "end_offset": 7, "type": "<IDEOGRAPHIC>", "position": 6},
  {"token": "的", "start_offset": 7, "end_offset": 8, "type": "<IDEOGRAPHIC>", "position": 7},
  {"token": "笔", "start_offset": 8, "end_offset": 9, "type": "<IDEOGRAPHIC>", "position": 8},
  {"token": "记", "start_offset": 9, "end_offset": 10, "type": "<IDEOGRAPHIC>", "position": 9},
  {"token": "本", "start_offset": 10, "end_offset": 11, "type": "<IDEOGRAPHIC>", "position": 10},
  {"token": "厂", "start_offset": 11, "end_offset": 12, "type": "<IDEOGRAPHIC>", "position": 11},
  {"token": "商", "start_offset": 12, "end_offset": 13, "type": "<IDEOGRAPHIC>", "position": 12}
]
}

```

从中可以看出,结果就是一个汉字变成了一个词了,基本上不能使用。

2. IKAnalyzer

目前 GitHub 上最新的版本只支持 Elasticsearch 2.1.1, 路径为 <https://github.com/medcl/elasticsearch-analysis-ik>。但现在最新的 Elasticsearch 已经到 2.3.0 了所以要经过处理一下才能支持:

1) 下载源码, 下载完后解压到任意目录, 然后修改 elasticsearch-analysis-ik-master 目录下的 pom.xml 文件。找到 <elasticsearch.version> 行, 然后把后面的版本号修改成 2.3.0。

2) 编译代码 mvn package。

3) 编译完成后会在 target/releases 生成 elasticsearch-analysis-ik-1.7.0.zip 文件。

4) 解压文件到 Elasticsearch/plugins 目录下。

5) 修改配置文件增加一行: index.analysis.analyzer.ik.type: "ik"。

6) 重启 Elasticsearch。

测试: 和上面的请求一样, 只是把分词替换成 ik。

返回值:

```

{
  "tokens": [
    {"token": " 联想 ", "start_offset": 0, "end_offset": 2, "type": "CN_WORD", "position": 0},
    {"token": "全球", "start_offset": 3, "end_offset": 5, "type": "CN_WORD", "position": 1},
    {"token": "最大", "start_offset": 5, "end_offset": 7, "type": "CN_WORD", "position": 2},
    {"token": "笔记本", "start_offset": 8, "end_offset": 11, "type": "CN_WORD", "position": 3},
    {"token": "笔记", "start_offset": 8, "end_offset": 10, "type": "CN_WORD", "position": 4},
    {"token": "笔", "start_offset": 8, "end_offset": 9, "type": "CN_WORD", "position": 5},
    {"token": "记", "start_offset": 9, "end_offset": 10, "type": "CN_CHAR", "position": 6},
    {"token": "本厂", "start_offset": 10, "end_offset": 12, "type": "CN_WORD", "position": 7},
    {"token": "厂商", "start_offset": 11, "end_offset": 13, "type": "CN_WORD", "position": 8}
  ]
}

```

从中可以看出，两个分词器分词的结果还是有区别的。

IKA nanlyzer 有一个扩展词库，在 config\ik\custom 下在 mydict.dic 中增加需要的词组，然后重启 Elasticsearch。需要注意的是文件编码是 UTF-8 无 BOM 格式编码。

比如增加了赛克蓝德单词。然后再次查询：

请求：POST http://127.0.0.1:9200/_analyze/

参数：

```
{
  "analyzer": "ik",
  "text": "赛克蓝德是一家数据安全公司"
}
```

返回值：

```
{
  "tokens": [
    {"token": "赛克蓝德", "start_offset": 0, "end_offset": 4, "type": "CN_WORD", "position": 0},
    {"token": "克", "start_offset": 1, "end_offset": 2, "type": "CN_WORD", "position": 1},
    {"token": "蓝", "start_offset": 2, "end_offset": 3, "type": "CN_WORD", "position": 2},
    {"token": "德", "start_offset": 3, "end_offset": 4, "type": "CN_CHAR", "position": 3},
    {"token": "一家", "start_offset": 5, "end_offset": 7, "type": "CN_WORD", "position": 4},
    {"token": "一", "start_offset": 5, "end_offset": 6, "type": "TYPE_CNUM", "position": 5},
    {"token": "家", "start_offset": 6, "end_offset": 7, "type": "CN_WORD", "position": 6},
    {"token": "数据", "start_offset": 7, "end_offset": 9, "type": "CN_WORD", "position": 7},
    {"token": "安全", "start_offset": 9, "end_offset": 11, "type": "CN_WORD", "position": 8},
    {"token": "公司", "start_offset": 11, "end_offset": 13, "type": "CN_WORD", "position": 9}
  ]
}
```

从上面的结果可以看出已经支持赛克蓝德单词了。

7.3 插件

Elasticsearch 通过插件来增强其个性化功能需求。插件可以添加自定义映射类型，自定义分词器、本地脚本、自定义发现等。在 Elasticsearch 中，有三种类型的插件：

- ❑ **Java 插件**：这些插件只包含 jar 文件，并且必须安装在集群中的每个节点上。安装后，每个节点必须重新启动后才生效。
- ❑ **站点插件**：这些插件包含了静态的 Web 内容，如 JavaScript、HTML 和 CSS 文件，站点插件只需要在一个节点上安装，并且不需要重新启动就可以使用。站点插件通过 URL 进行访问，例如：[http://127.0.0.1:9200/_plugin/\[plugin name\]](http://127.0.0.1:9200/_plugin/[plugin name])。
- ❑ **混合插件**：混合插件是既包含 jar 文件又包含站点插件。

7.3.1 插件管理

插件是通过脚本来进行管理的，可以安装插件、查询插件和删除插件。正常情况下插件位于 \$ES_HOME/bin 下，通过 rpm 等安装的位置可能会不同。可以通过命令获取插件帮助：在目录下执行 plugin -h：

```
plugin -h
NAME
  plugin - Manages plugins
SYNOPSIS
  plugin <command>
DESCRIPTION
  Manage plugins
COMMANDS
  install Install a plugin
  remove Remove a plugin
  list List installed plugins
NOTES
  [*] For usage help on specific commands please type "plugin <command> -h"
```

注意：Linux 需要用管理员来运行，比如 sudo plugin -h。

7.3.2 插件安装

每种插件都会有插件安装文档，但大多数情况下，插件安装有以下几种方式：

□ 核心插件。

```
sudo bin/plugin install [plugin_name]
```

例如安装 Lucene icu 插件：

```
sudo bin/plugin install analysis-icu
```

这样就会按照合适的版本安装到 Elasticsearch 中。

□ 非核心插件。非核心插件可以是官方提供的，也可以是社区提供的，可以从官方 Maven 或者 GitHub 中下载安装。

```
sudo bin/plugin install [org]/[user|component]/[version]
```

例如，安装 GitHub 上的插件：

```
plugin install lmenezes/elasticsearch-kopf
```

插件会尝试先到官方去下载，如果没有找到会到 maven.com 中去下载，如果再没有找到回到 GitHub 中去下载。脚本还是非常智能的。

当我们直接从 Maven 中央库中安装时可以使用下面的方式，最后的版本号是必须要写的：

```
plugin install org.elasticsearch.plugin/mapper-attachments/3.0.0
```

□ 从自定义网址或者本地安装。例如，在本地文件系统中安装一个插件，可以运行：

```
plugin install file:///path/to/plugin.zip。
```

插件脚本会拒绝从一个不受信任的证书的 HTTPS 网站安装。使用 HTTPS 证书自签名，需要添加 CA 证书到本地信任库，可以通过 Java 脚本添加：

```
plugin -Djavax.net.ssl.trustStore=/path/to/trustStore.jks install https://...
```

❑ 查询插件：plugin list

❑ 删除插件：plugin remove [pluginname]，jar 插件删除后要重启。

❑ Silent/Verbose 参数：当使用 -verbose 参数输出更多的调试信息，当使用 -silent 参数时可以关闭所有输出。该脚本可以返回下面的代码：

0：一切都好

64：未知命令或不正确的选项参数

74：输入输出错误

70：任何其他错误

❑ 自定义配置目录：如果 elasticsearch.yml 配置文件在一个自定义的位置，在使用插件脚本时需要指定配置文件的路径，例如：

```
plugin -Des.path.conf=/path/to/custom/config/dir install <plugin name>
```

可以设置 conf_dir 环境变量来指定自定义配置文件的路径。

❑ 超时设置：可以指定超时时间，例如等待 30 秒：

```
plugin install mobz/elasticsearch-head -timeout 30s
```

当设置为 0 时一直等待。

❑ 代理设置：可以通过代理安装插件，Java 的设置是 proxyHost 和 proxyPort。例如：
在 Unix 中：

```
plugin install mobz/elasticsearch-head -DproxyHost=host_name -DproxyPort=port_number
```

在 Windows 中，需要添加参数到 java_opts 环境变量，例如：

```
set JAVA_OPTS="-DproxyHost=host_name -DproxyPort=port_number"
```

```
plugin install mobz/elasticsearch-head
```

❑ 自定义插件目录：插件目录可以在配置文件 elasticsearch.yml 中进行修改，参数：
path.plugins: /path/to/custom/plugins/dir。

❑ 强制插件：可以在配置文件 elasticsearch.yml 中添加 plugin.mandatory 来设置必须要安装的插件，例如：plugin.mandatory: mapper-attachments,lang-python。注意：出于安全的原因，如果缺少一个强制插件，该节点将不会启动成功。

7.3.3 插件清单

在 Elasticsearch 中有非常多的插件，这些插件主要分成以下几类：API 插件、报警插件、

分词插件、发现插件、管理和站点插件、映射器插件、脚本插件、安全插件、快照 / 恢复插件、传输插件。下面介绍其中几个。

1. API 插件

API 插件主要对 Elasticsearch 添加新的 API 特性或者功能，通常用于搜索或者映射。

核心插件是 delete-by-query。该插件可以通过查询来删除文档，在内部，它使用 Scroll API 和 Bulk API 来删除文件。注意不要用此方法来删除大量的文档，因为它是一条一条的删除文档，这会导致消耗的时间比较长，比较好的方式是创建一个新的索引，把需要的文档 copy 过去。

使用方法：

请求：DELETE http://127.0.0.1:9200/secisland/secilog/_query?q=user:kimchy

或者：

请求：DELETE http://127.0.0.1:9200/secisland/secilog/_query

参数：

```
{
  "query": { "term": { "user": "kimchy" } }
}
```

社区插件包括如下几个：

- carrot2 Plugin：基于 carrot2 或者 Lingo3G 算法的聚合插件。
- SQL language Plugin：通过 SQL 语法来搜索 Elasticsearch。
- WebSocket Change Feed Plugin：通过 WebSocket 连接到 Elasticsearch 节点和接收数据。

2. 报警插件

当 Elasticsearch 的索引等指标超出阈值时报警插件会触发报警。Watcher 插件是 Elasticsearch 官方支持的报警插件，但是这个插件是付费的，初次安装有 30 天的试用期，过期后如果想继续使用可以付费购买。

3. 分词插件

核心插件：

analysis-icu：ICU 分析插件集成 Lucene ICU 模块到 Elasticsearch，它支持 Unicode 的 ICU 库，包括更好的分析亚洲语言。

analysis-phonetic：语音分析插件。

analysis-smartcn：中科院的中文分词插件，不支持扩展，不建议使用。

analysis-stempel：支持 Lucene 的 Stempel analysis 模块的插件。

社区插件：

IK Analysis Plugin：比较好的中文插件，见前面 7.2 节的介绍。

Mmseg Analysis Plugin：集成 Lucene mmseg4j-analyzer 的插件，支持中文。

Pinyin Analysis Plugin: 集成 Pinyin4j 插件, Pinyin4j 是支持汉字和拼音系统之间转换的最受欢迎流行的 java 库。可自定义拼音输出格式。

Network Addresses Analysis Plugin: 网络 MAC 地址分析插件。

4. 发现插件

发现节点插件是替换 Elasticsearch 自身发现功能的插件。

核心插件主要包括:

AWS Cloud: 亚马逊云。

Azure Cloud: 微软云。

GCE Cloud: Google 云。

Multicast: 组播插件发送多播消息, 发现在集群中的其他节点。

社区插件包括:

eseka Discovery Plugin: 支持 Gossip 协议的 Akka Cluster 集群节点。

Kubernetes Discovery Plugin: 通过 Kubernetes API 发现节点。

5. 管理和站点插件

Marvel 是官方支持的管理插件, 是收费插件。

社区插件是 Elasticsearch Head Plugin: mobz/elasticsearch-head 插件, 推荐使用。

7.4 小结

本章介绍了 Elasticsearch 使用分词器从文档中提取出若干词元来支持索引的存储和搜索。列举出了内置的分词器和自定义分词器的方法。针对中文分词采取特殊的分词方式, 并且介绍了可用的中文分词器插件。

7.3.3 插件清单



第8章 Chapter 8

高级配置

8.1.3 网络配置

Elasticsearch 默认绑定本地网络接口。在 Elasticsearch 中，网络配置是用于配置 Elasticsearch 的网络设置。本章主要介绍 Elasticsearch 的网络配置。本章主要介绍的组件配置有：

8.1 HTTP 相关配置

在 Elasticsearch 的配置中，主要有两种配置方式，一种是静态配置，另一种是动态配置。静态配置参数只能在配置文件中事先写好，动态配置参数可以通过 `_cluster/settings` 进行设置。本章主要介绍的组件配置有：

网络配置，包括本地网关、HTTP、网络、传输等配置。脚本配置，脚本模块的配置。快照和恢复配置，快照备份恢复模块的配置。线程池配置，线程池模块的配置。索引配置，全局索引相关设置模块的配置。

8.1 网络相关配置

Elasticsearch 在设计的时候就考虑到了分布式，这些分布式的基础都是来源于网络；包括 Elasticsearch 的人机交互接口 HTTP REST API 也是基于网络的。本章重点介绍 Elasticsearch 网络相关的配置。

8.1.1 本地网关配置

本地网关是当所有集群重新启动时存储集群状态和分片数据的模块。

下面的静态设置控制集群中每个数据节点在恢复任何本地存储的分片之前，应该等待多长时间：

- `gateway.expected_nodes`——集群中预计节点（数据或主节点）的数量。本地分片恢复会在预计数量节点加入集群之后开始执行。默认值为 0。
- `gateway.expected_master_nodes`——集群中主节点预计数量。本地分片恢复会在预计数量主节点加入集群之后开始执行。默认值为 0。

- `gateway.expected_data_nodes`——集群中数据节点预计数量。本地分片恢复会在预计数量数据节点加入集群之后开始执行。默认值为 0。
- `gateway.recover_after_time`——如果没有达到预计节点数量，本地分片在尝试恢复之前会等待配置的时间量。默认值为 5m。

8.1.2 HTTP 配置

HTTP 模块允许通过 HTTP 来调用 Elasticsearch 接口。

HTTP 机制本质上是完全异步的，意味着等待响应的线程不会被阻塞。

HTTP 的配置不会动态更新，所以应该在 `elasticsearch.yml` 中进行配置来产生影响，HTTP 的相关配置参见表 8-1。

表 8-1 HTTP 相关配置

设置	说明
<code>http.port</code>	绑定端口范围，默认为 9200-9300
<code>http.publish_port</code>	与此节点通信时，HTTP 客户端应该使用的端口。当一个集群节点在代理或防火墙之后，而且外部无法直接访问 <code>http.port</code> ，默认实际端口通过 <code>http.port</code> 赋值
<code>http.bind_host</code>	绑定 HTTP 服务的主机地址。默认为 <code>http.host</code> 或者 <code>network.bind_host</code> 参数值。
<code>http.publish_host</code>	发布给 HTTP 客户端用来连接的主机地址。默认为 <code>http.host</code> 或 <code>network.publish_host</code> 参数值
<code>http.host</code>	用于设置 <code>http.bind_host</code> 和 <code>http.publish_host</code>
<code>http.max_content_length</code>	HTTP 请求的最大内容。默认是 100MB。如果设置大于 <code>Integer.MAX_VALUE</code> ，会重置为 100MB
<code>http.max_initial_line_length</code>	HTTP URL 的最大长度。默认值为 4MB
<code>http.max_header_size</code>	请求头的最大长度。默认值为 8MB
<code>http.compression</code>	是否支持压缩。默认值为 <code>false</code>
<code>http.compression_level</code>	定义使用的压缩级别。默认值为 6
<code>http.cors.enabled</code>	启用或禁用跨源资源共享。默认是 <code>false</code>
<code>http.cors.allow-origin</code>	允许的请求源，默认拒绝所有源。如果预先添加 / 值，会被作为正则表达式。可以支持 HTTP 和 HTTPS，例如 <code>/https?:\V/127.0.0.1(:[0-9]+)?/</code> 。* 会被作为一个安全风险而不是对任何请求源允许跨源请求
<code>http.cors.max-age</code>	浏览器发送一个“预检”可选请求来决定 CORS 设置。 <code>max-age</code> 定义了结果需要缓存的时间。默认是 1728000（20 天）
<code>http.cors.allow-methods</code>	允许的请求方法。默认选项有：HEAD、GET、POST、PUT、DELETE
<code>http.cors.allow-headers</code>	允许的请求头。默认为 X-Requested-With、Content-Type、Content-Length
<code>http.cors.allow-credentials</code>	是否 Access-Control-Allow-Credentials 请求头应该被返回。当设置为 <code>true</code> ，仅返回这个请求头，默认为 <code>false</code>
<code>http.detailed_errors.enabled</code>	启用或禁用响应输出中的详细错误信息和堆栈跟踪输出。注意：当设置为 <code>false</code> ，而且指定了 <code>error_trace</code> 请求参数，会返回一个错误；当 <code>error_trace</code> 没有指定，会返回一个简单的信息。默认值为 <code>true</code>
<code>http.pipelining</code>	启用或禁用 HTTP 管道，默认值为 <code>true</code>
<code>http.pipelining.max_events</code>	HTTP 连接关闭之前内存中排队事件的最大数量。默认值为 10000

禁用 HTTP

HTTP 模块可以通过设置 `http.enabled` 为 `false` 来完全禁用。Elasticsearch 节点（以及 Java 客户端）使用传输接口进行内部通信而不是 HTTP 接口，因此禁用 HTTP 接口不会影响节点间的数据传输，可以在数据节点禁用 HTTP。

8.1.3 网络配置

Elasticsearch 默认绑定本地网络地址。足够运行一个本地开发服务器（或者在同一机器上运行多个节点进行集群开发）。但是为了在多个服务器上运行一个真正的产品集群，需要配置一些基础的网络配置。



警告 不要向公网暴露未经保护的节点。

1. 常用网络配置

常用的网络配置见表 8-2。

表 8-2 常用网络配置选项

参数	说明
<code>network.host</code>	节点会绑定这个主机名或 IP 地址并且发布（广播）到集群中的其他节点。接受 IP 地址、主机名、一个特殊值（下面将进行详述）或者这些类型的任意组合数组。默认值为 <code>_local_</code>
<code>discovery.zen.ping.unicast.hosts</code>	为了加入一个集群，节点至少需要知道一些集群中其他节点的主机名或 IP 地址。这个设置提供节点尝试连接的其他节点的最初列表。接受 IP 地址或主机名。默认值为 <code>["127.0.0.1", "::1"]</code>
<code>http.port</code>	绑定接收 HTTP 请求的端口。接受单独的值或者一个范围。如果指定了一个范围，节点会绑定范围中第一个可用的端口。默认为 9200-9300
<code>transport.tcp.port</code>	绑定节点间通信的端口。接受单独的值或者一个范围。如果指定了一个范围，节点会绑定范围中第一个可用的端口。默认为 9300-9400

其中，`network.host` 的特殊值有：

- ☐ `_[networkInterface]`——网络接口地址，例如 `_en0_`。
- ☐ `_local_`——系统中的任何回环地址，例如 `127.0.0.1`。
- ☐ `_site_`——系统中的任何本地站点地址，例如 `198.168.0.1`。
- ☐ `_global_`——系统中的任何广域地址，例如 `8.8.8.8`。

这些特殊值默认支持 IPv4 和 IPv6，但是也可以使用 `:ipv4` 或 `:ipv6` 标识符进行限定。例如，`_en0:ipv4_` 仅会绑定 `en0` 接口的 IPv4 地址。

2. 高级网络配置

通用网络配置中的 `network.host` 是同时设置绑定主机和发布主机的快捷方式。在高级应用的情况下，比如在代理服务器后面运行的时候，需要使用高级网络配置：

❑ `network.bind_host`——指定节点应该绑定的网络接口来监听传入请求。节点可以绑定多个接口，例如两块网卡，或者一个本地站点地址和本地地址。默认为 `network.host` 的值。

❑ `network.publish_host`——发布主机是节点发布到集群中其他节点的单独接口，其他节点可以连接到本节点。节点可以绑定多个地址，但是只发布一个地址。如果不进行指定，默认是 `network.bind_host` 中的“最佳”地址，IPv4/IPv6 堆栈的最佳排序，然后根据可达性进行选择。

这些设置可以接受 `network.host` 同样的配置，即接受 IP 地址、主机名和特殊值。

3. 高级 TCP 配置

任何使用 TCP 协议的组件（比如 HTTP 和传输模块）共享这些设置：

❑ `network.tcp.no_delay`——启用或禁用 TCP 无延迟设置。默认为 `true`。

❑ `network.tcp.keep_alive`——启用或禁用 TCP 保持连接。默认是 `true`。

❑ `network.tcp.reuse_address`——地址是否应该被重用。在非 Windows 机器上默认是 `true`。

❑ `network.tcp.end_buffer_size`——TCP 发送缓存的大小（指定单位）。默认没有显式设置。

❑ `network.tcp.receive_buffer_size`——TCP 接收缓存的大小（指定单位）。默认没有显式设置。

4. 传输和 HTTP 协议

Elasticsearch 节点公开两种网络协议，继承上面的配置。但也可以进一步进行独立配置：

❑ TCP 传输——用于集群中节点间的通信。详见下节“传输配置”。

❑ HTTP——详见上一节“HTTP 配置”。

8.1.4 传输配置

传输模块用于集群中的节点间进行内部通信。每个请求使用传输模块从一个节点发送到另一个节点中（例如当一个 HTTP GET 请求在一个节点上执行，实际的执行过程应该在另一个包含数据的节点上进行）。

传输机制本质上是异步的，意味着线程不会被阻塞等待响应。

1. TCP 传输

TCP 传输是传输模块使用 TCP 协议的一种实现，其设置参见表 8-3：

表 8-3 TCP 传输设置

设置	描述
<code>transport.tcp.port</code>	绑定的端口范围。默认为 9300-9400
<code>transport.publish_port</code>	集群中其他节点与本节点通信应该使用的端口

(续)

设置	描述
transport.bind_host	传输服务绑定的主机地址。默认是 transport.host (如果设置) 或 network.bind_host
transport.publish_host	发布给集群中的节点用来连接的主机地址。默认为 transport.host (如果设置) 或 network.publish_host
transport.host	用来设置 transport.bind_host 和 transport.publish_host, 默认为 transport.host 或 network.host
transport.tcp.connect_timeout	端口连接超时设置 (使用时间设置格式)。默认为 30s
transport.tcp.compress	设置为 true 来启用节点压缩 (LZF)。默认为 false
transport.ping_schedule	安排规律的 ping 消息来确保连接的保持。传输客户端默认为 5s, 其他地方为 -1 (禁用)

通过使用传输配置文件, Elasticsearch 可以对不同的接口绑定多个端口。示例配置如下:

- ☐ transport.profiles.default.port: 9300-9400
- ☐ transport.profiles.default.bind_host: 10.0.0.1
- ☐ transport.profiles.client.port: 9500-9600
- ☐ transport.profiles.client.bind_host: 192.168.0.1
- ☐ transport.profiles.dmz.port: 9700-9800
- ☐ transport.profiles.dmz.bind_host: 172.16.1.2

2. 本地传输

这是个方便的传输用于在 JVM 中执行集成测试。当使用 NodeBuilder#local(true), 会自动启用本地传输。

3. 传输追踪

传输模块拥有专门的追踪日志, 记录到达和发出的请求。通过将 transport.tracer 设置为 TRACE, 可以动态激活日志, 如下所示:

请求: PUT http://127.0.0.1:9200/_cluster/settings

```
{
  "transient" : {"logger.transport.tracer" : "TRACE"}
}
```

使用包含和排除设置, 可以控制需要追踪的行为, 如下所示:

请求: PUT http://127.0.0.1:9200/_cluster/settings

```
{
  "transient" : {"transport.tracer.include" : "*"
    "transport.tracer.exclude" : "internal:discovery/zen/fd*"}
}
```

8.2 脚本配置

脚本模块可以使用脚本对 Elasticsearch 的字段进行再次处理。例如，可以用来重新评估查询的自定义得分，可以对索引中的某个字段再次加工处理。

脚本默认是关闭的，如果这个时候执行脚本会报以下错误：

```
{
  "error": {
    "root_cause": [
      {
        "type": "remote_transport_exception",
        "reason": "[Left-Winger] [127.0.0.1:9300] [indices:data/write/update[s]]"
      }
    ],
    "type": "illegal_argument_exception",
    "reason": "failed to execute script",
    "caused_by": {
      "type": "script_exception",
      "reason": "scripts of type [indexed], operation [update] and lang [groovy]
        are disabled"
    }
  },
  "status": 400
}
```

可以通过配置来启用脚本引擎，配置的位置在 `elasticsearch.yml` 文件中添加如下内容：

```
script.inline: true
script.indexed: true
```

对这些设置，有三种配置值：

- ☐ `false`——完全禁用脚本。
- ☐ `true`——启用脚本。
- ☐ `sandbox`——脚本仅可以用沙盒语言执行。

默认配置值为：

```
script.inline: sandbox
script.indexed: sandbox
script.file: true
```

也可以在以下操作中执行脚本：

- ☐ `aggs`——聚合
- ☐ `search`——搜索接口、过滤接口或建议接口（例如 `filters`、`script_fields`）。
- ☐ `update`——升级接口。
- ☐ `plugin`——通用 `plugin` 类别下，脚本使用的任何插件。

插件也可以进行自定义操作，利用这种格式：`${pluginName}_${operation}`。例如在任何脚本引擎上禁用更新和映射：

```
script.update: false
script.mapping: false
```

也支持明确的脚本语言设置，需要 `script.engine.<engine>` 前缀，优先权高于其他通用设置：

```
script.engine.groovy.file.aggs: true
script.engine.groovy.file.mapping: true
script.engine.groovy.file.search: true
script.engine.groovy.file.update: true
script.engine.groovy.file.plugin: true
script.engine.groovy.indexed.aggs: true
script.engine.groovy.indexed.mapping: false
script.engine.groovy.indexed.search: true
script.engine.groovy.indexed.update: false
script.engine.groovy.indexed.plugin: false
script.engine.groovy.inline.aggs: true
script.engine.groovy.inline.mapping: false
script.engine.groovy.inline.search: false
script.engine.groovy.inline.update: false
script.engine.groovy.inline.plugin: false
```

系统的脚本模块默认使用 Groovy 作为脚本语言。可以通过设置 `script.default_lang` 进行修改。

可以用语言插件来支持不同的语言脚本。所以在使用脚本接口的参数中提供了 `lang` 参数定义脚本语言，见表 8-4。

表 8-4 脚本语言插件

语言	沙盒	必要插件
Groovy	否	内置
Expression	是	内置
Mustache	是	内置
JavaScript	否	elasticsearch-lang-javascript
Python	否	elasticsearch-lang-python

为了增加安全性，Elasticsearch 不允许在请求中指定非沙盒语言。默认的脚本目录位置可以在 `elasticsearch.yml` 中设置 `path.scripts` 进行修改。放置在这个目录中的脚本会自动选中并且可以被使用。一旦脚本放在这个目录中，可以通过脚本名进行引用。

8.2.1 脚本使用

在 Elasticsearch 中使用脚本有三种方式：

- ❑ 直接在请求体中使用脚本。
- ❑ 把脚本存储在索引中，通过引用脚本 id 来使用。
- ❑ 把脚本存储在本地磁盘中，默认的位置为：`elasticsearch/config/scripts`，通过引用脚

本名称进行使用。

下面通过举例来说明使用脚本的三种方法。

首先建一个索引，添加一条数据：

请求：PUT http://127.0.0.1:9200/secisland/secilog/1

```
{ "eventCount": 1, "eventName": "linux login event" }
```

下面我们用脚本对 eventCount 做加法操作。

用第一种方法直接在请求中执行脚本：

请求：POST http://127.0.0.1:9200/secisland/secilog/1/_update

```
{
  "script": "ctx._source.eventCount+=count",
  "params": { "count": 4 }
}
```

下面我们用第二种方法操作，在用第二种方法前，先要把脚本存储在 Elasticsearch 中：

请求：POST http://127.0.0.1:9200/_scripts/groovy/indexedCalculateCount

```
{ "script": "ctx._source.eventCount+=count" }
```

然后通过脚本 id 进行文档操作：

请求：POST http://127.0.0.1:9200/secisland/secilog/1/_update

```
{
  "script": {
    "id": "indexedCalculateCount",
    "lang": "groovy",
    "params": { "count": 8 }
  }
}
```

下面，我们用第三种方法操作，在第三种方法操作前，先要把脚本存储在文件中，文件名为 indexedCalculateCount.groovy，文件中的内容为：ctx._source.eventCount+=count。



注意 Elasticsearch 对文件读取有个时间，刚建好后，不能生效，做验证的时候可以重启进行生效。

请求：POST http://127.0.0.1:9200/secisland/secilog/1/_update/

```
{
  "script": {
    "file": "indexedCalculateCount",
    "lang": "groovy",
    "params": { "count": 8 }
  }
}
```

8.2.2 脚本配置

1. 索引脚本

Elasticsearch 可以在名为 `_scripts` 的内部索引中存储脚本，并且通过 `id` 进行引用。脚本请求的格式：`/_scripts/{lang}/{id}`。

`lang` 表示脚本语言，`id` 表示脚本编号。

保存脚本的示例如下：

请求：POST `http://127.0.0.1:9200/_scripts/groovy/indexedCalculateScore`

```
{"script": "log(_score * 2) + my_modifier"}
```

使用脚本的示例如下：

请求：POST `http://127.0.0.1:9200/_search`

```
{
  "query": {
    "function_score": {
      "query": {
        "match": {"body": "foo"}
      },
      "functions": [
        {
          "script_score": {
            "script": {
              "id": "indexedCalculateScore", // 脚本名称要和存储中的名称一致
              "lang": "groovy",
              "params": {"my_modifier": 8}
            }
          }
        }
      ]
    }
  }
}
```

查看脚本如下：

请求：GET `http://127.0.0.1:9200/_scripts/groovy/indexedCalculateScore`

删除脚本如下：

请求：DELETE `http://127.0.0.1:9200/_scripts/groovy/indexedCalculateScore`

2. 启用动态脚本

在应用或代理后面执行 Elasticsearch，可以从外界保护 Elasticsearch。如果允许用户运行内联脚本或索引脚本，会继承运行 Elasticsearch 的用户权限。因此动态脚本默认仅支持沙盒语言。

首先，应该使用 `root` 用户权限运行 Elasticsearch，可以允许脚本在服务器上访问或做任

何事情。其次，不应该让用户直接访问 Elasticsearch，而是要有一个中间代理应用。如果确实想要用户直接访问 Elasticsearch，需要决定是否足够信任用户来运行脚本。

3. 脚本自动重载

周期性扫描 config/scripts 目录的修改。新的和修改的脚本会被重载，删除的脚本会从预加载脚本缓存中移除。重载频率可以使用 resource.reload.interval 设置指定，默认值为 60s。设置 script.auto_reload_enabled 为 false 可以完全禁用脚本重载。

4. 本地 (Java) 脚本

有时，groovy 和 expressions 是不够的。在这种情况下，可以执行本地脚本。

执行本地脚本的最好方式是编写并且安装一个插件。

为了注册实际脚本，需要引用 NativeScriptFactory 来构成脚本。实际脚本需要继承 AbstractExecutableScript 或 AbstractSearchScript。第二种可能是最有用的并且有多个可以继承的子类，比如 AbstractLongSearchScript、AbstractDoubleSearchScript 和 AnstractFloatSearchScript。最后，插件需要通过 onModule(ScriptModule) 方法注册本地脚本。

```
public class MyNativeScriptPlugin extends Plugin {
    @Override
    public String name() {
        return "my-native-script";
    }
    @Override
    public String description() {
        return "my native script that does something great";
    }
    public void onModule(ScriptModule scriptModule) {
        //my_script 名称为本地 Java 脚本名称
        scriptModule.registerScript("my_script", MyNativeScriptFactory.class);
    }
}

public static class MyNativeScriptFactory implements NativeScriptFactory {
    //MyNativeScriptFactory 返回的脚本类是 MyNativeScript 类
    @Override
    public ExecutableScript newScript(@Nullable Map<String, Object> params) {
        return new MyNativeScript();
    }
    @Override
    public boolean needsScores() {
        return false;
    }
}

// 真正执行的脚本代码
public static class MyNativeScript extends AbstractFloatSearchScript {
    @Override
    public float runAsFloat() {
        float a = (float) source().get("a");
        float b = (float) source().get("b");
```

```

    return a * b;
  }
}

```

可以执行脚本通过指定 lang 为 native, 指定 inline 为脚本名:

请求: POST http://127.0.0.1:9200/_search

```

{
  "query": {
    "function_score": {
      "query": {
        "match": {"body": "foo"}
      },
      "functions": [
        {
          "script_score": {
            "script": {
              "inline": "my_script", // 脚本的名称
              "lang": "native"
            }
          }
        }
      ]
    }
  }
}

```

5. Lucene 表达式脚本

Lucene 的表达式模块提供了一个机制来将 Javascript 表达式编译成字节码。表达式脚本可以用于 script_score、script_fields、排序脚本和数字型聚合脚本。

表达式脚本变量可以接受:

- ☐ 单值文档字段, 例如 doc['myfield'].value, 也可以写作 doc['myfield']。
- ☐ 传给脚本的参数, 例如 mymodifier。
- ☐ 当前文档的得分, _score (只有用在 script_score 的时候)。

表达式脚本日期类型可以使用这些方法:

- ☐ getYear()
- ☐ getMonth()
- ☐ getDayOfMonth()
- ☐ getHourOfDay()
- ☐ getMinutes()
- ☐ getSeconds()

比如, 获取日期字段间年份的不同:

```
doc['date1'].getYear() - doc['date0'].getYear()
```

相对于其他脚本语言，有一些限制：

- ☐ 只接受数字型字段。
- ☐ 保存的字段不可用。
- ☐ 如果只有一部分文档包含字段值，缺失字段值的文档默认值为 0。

6. 得分

聚合中所有可以使用的脚本，当前文档的得分可以用 `_score` 获得。

7. 文档字段

大多数脚本围绕指定文档字段数据来使用。`doc['field_name']` 可以用来访问文档内指定字段数据。注意，只能是简单值字段（不能返回 `Json` 对象）并且只有不分词字段或单索引词字段是有意义的。

可以从字段中获取数据，见表 8-5。

表 8-5 脚本提取数据

表达式	描述
<code>doc['field_name'].value</code>	字段的本地值
<code>doc['field_name'].values</code>	字段的本地数组值。如果字段没有值，返回一个空数组
<code>doc['field_name'].empty</code>	布尔值，表示文档中的字段是否有值
<code>doc['field_name'].multiValued</code>	布尔值，表示文档中有多个值的字段
<code>doc['field_name'].lat</code>	地理点类型的纬度
<code>doc['field_name'].lon</code>	地理点类型的经度
<code>doc['field_name'].lats</code>	地理点类型的纬度（复数）
<code>doc['field_name'].lons</code>	地理点类型的经度（复数）
<code>doc['field_name'].distance(lat,lon)</code>	地理点字段和提供的经纬度之间的平面距离（米）
<code>doc['field_name'].distanceWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的平面距离（米）
<code>doc['field_name'].distanceInMiles (lat,lon)</code>	地理点字段和提供的经纬度之间的平面距离（英里）
<code>doc['field_name'].distanceInMilesWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的平面距离（英里）
<code>doc['field_name'].distanceInKm (lat,lon)</code>	地理点字段和提供的经纬度之间的平面距离（千米）
<code>doc['field_name'].distanceInkmWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的平面距离（千米）
<code>doc['field_name'].arcDistance (lat,lon)</code>	地理点字段和提供的经纬度之间的天穹距离（米）
<code>doc['field_name'].arcDistanceWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的天穹距离（米）
<code>doc['field_name'].arcDistanceInMiles (lat,lon)</code>	地理点字段和提供的经纬度之间的天穹距离（英里）
<code>doc['field_name'].arcDistanceInMilesWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的天穹距离（英里）
<code>doc['field_name'].arcDistanceInKm (lat,lon)</code>	地理点字段和提供的经纬度之间的天穹距离（千米）

(续)

表达式	描述
<code>doc['field_name'].arcDistanceInKmWithDefault(lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的天穹距离(千米)
<code>doc['field_name'].factorDistance (lat,lon)</code>	地理点字段从提供的经纬度之间的距离因子
<code>doc['field_name'].factorDistance (lat,lon,default)</code>	地理点字段从提供的经纬度和默认值的距离因子
<code>doc['field_name'].geohashDistance (geohash)</code>	地理点字段从提供的地理散列的天穹距离(米)
<code>doc['field_name'].geohashDistanceInKm (geohash)</code>	地理点字段从提供的地理散列的天穹距离(千米)
<code>doc['field_name'].geohashDistanceInMiles (geohash)</code>	地理点字段从提供的地理散列的天穹距离(英里)

8. 保存的字段

执行脚本的时候,保存的字段也可以访问。通过使用 `_fields['my_field_name'].value` 或 `_fields['my_field_name'].values`。

9. 在脚本中访问文档的得分

当使用脚本计算文档得分(例如,通过 `function_score` 查询),可以在 `groovy` 脚本中使用 `_score` 变量访问得分。

10. 源字段

当执行脚本的时候,可以访问源字段。每个文档都会加载源字段、分析并且提供给脚本进行评估。使用方式,形如 `_source.obj1.obj2.field3`。

11. Groovy 内置方法

见表 8-6。

表 8-6 Groovy 内置方法

方法	描述
<code>sin(a)</code>	返回角度的三角正弦
<code>cos(a)</code>	返回角度的三角余弦
<code>tan(a)</code>	返回角度的三角正切
<code>asin(a)</code>	返回反正弦值
<code>acos(a)</code>	返回反余弦值
<code>atan(a)</code>	返回反正切值
<code>toRadians(angdeg)</code>	转换角度度量近似为弧度
<code>toDegrees(angrad)</code>	转换弧度度量近似为角度
<code>exp(a)</code>	返回欧拉数 e 的指数值
<code>log(a)</code>	返回值的自然对数(基于 e)
<code>log10(a)</code>	返回值基于 10 的对数
<code>sqrt(a)</code>	返回值的平方根
<code>cbrt(a)</code>	返回双精度值的立方根

(续)

方法	描述
IEEEremainder(f1,f2)	进行余数运算
ceil(a)	向下取整
floor(a)	向上取整
rint(a)	取整
atan2(y,x)	转换直角坐标 (x,y) 到极坐标 (r,θ), 返回 θ 值
pow(a,b)	返回 a^b
round(a)	取整
random()	返回随机双精度数
abs(a)	返回值的绝对值
max(a,b)	返回大值
min(a,b)	返回小值
ulp(d)	返回参数的 ulp 的大小
signum(d)	返回自变量的符号函数
sinh(x)	返回值的双曲正弦值
cosh(x)	返回值的双曲余弦值
tanh(x)	返回值的双曲切线
hypot(x,y)	返回 $\sqrt{x^2+y^2}$

8.3 快照和恢复配置

在 Elasticsearch 中可以创建快照, 它创建单个索引或整个集群到一个远程的存储库中。在以前的版本中, 系统只支持共享存储的快照创建, 最新的版本可以通过插件支持更多的方式。在执行快照操作前, 需要在 Elasticsearch 中注册快照仓库。注册仓库的设置需要执行仓库类型请求, 例如:

请求: PUT http://127.0.0.1:9200/_snapshot/my_backup

参数:

```
{
  "type": "fs",
  "settings": { ... repository specific settings ... }
}
```

当注册后, 可以通过查询来得到之前注册的信息。当然系统也支持用逗号分开来查询多个注册, 支持通配符查询多个注册, 甚至可以使用 `_all` 查询所有注册信息。

请求: GET http://127.0.0.1:9200/_snapshot/my_backup

返回值:

```
{
```




```

"my_backup": {
  "type": "fs",
  "settings": {"compress": "true", "location": "/mount/backups/my_backup"}
}

```

共享文件系统存储 ("type": "fs") 使用共享文件系统来存储快照。为了注册共享的文件系统存储库，必须将同一个共享文件系统安装到所有主节点和数据节点的相同位置上。这个位置（或它的父目录）必须在所有主节点、数据节点上设置 `path.repo` 参数。假设共享文件系统安装在 `/mount/backups/my_backup` 目录下，在 `elasticsearch.yml` 文件中的设置应增加：

```
path.repo: ["/mount/backups", "/mount/longterm_backups"]
```

 **提示** `path.repo` 设置支持微软 Windows UNC 路径，只要服务器名和共享名称进行正确的设置，例如：

```
path.repo: [ "\\\\MY_SERVER\\Snapshots" ]
```

只有当所有的节点都重新启动后，使用以下的命令可以为名称为 `my_backup` 的共享文件系统库进行快照和恢复，例如：

请求：PUT `http://127.0.0.1:9200/_snapshot/my_backup`

参数：

```

{
  "type": "fs",
  "settings": {"location": "/mount/backups/my_backup", "compress": true}
}

```

系统支持相对路径的，如果存储位置指定为相对路径，则不需要写具体的全路径，例如：

请求：PUT `http://127.0.0.1:9200/_snapshot/my_backup`

参数：

```

{
  "type": "fs",
  "settings": {"location": "my_backup", "compress": true}
}

```

系统支持的参数如下所示：

□ `location`——快照位置，必须要有

□ `compress`——打开快照文件的压缩。压缩仅适用于元数据文件，数据文件不进行压缩。默认值为 `true`

□ `chunk_size`——如果需要可以把大的文件分解成不同的快照。块的大小可以指定字节例如 1G、10m、5K。默认值为 `null`，表示无限的块大小

- ❑ `max_restore_bytes_per_sec`——每个节点的恢复速度。默认值为 40MB 每秒
- ❑ `max_snapshot_bytes_per_sec`——每个节点生成的快照速度。默认值为 40MB 每秒
- ❑ `readonly`——使存储库只读。默认值为 `false`

1. 只读仓库

URL 仓库 (`"type": "url"`) 用于共享文件系统库中创建只读方式数据仓库。在 `url` 参数中指定指向共享的文件系统存储库的根节点，支持以下设置：

- ❑ `url`：强制性的快照位置。URL 参数支持以下协议：HTTP、HTTPS、FTP、File、jar，在 `http`、`https`、`ftp` 这些协议中，可以支持白名单，白名单的设置在于 `repositories.url.allowed_urls` 参数中，例如：

```
repositories.url.allowed_urls: ["http://www.example.org/root/*", "https://*.mydomain.com/*?#*"]
```

存储库可以通过插件支持更多的方式，比如：AWS 云插件的 S3 存储库、Hadoop 的 HDFS 插件环境、Azure 云存储库插件。

当注册了一个存储库时，它立即在所有主节点和数据节点上验证，以确保它在集群中的所有节点上都可以使用。验证参数可用于在注册或更新存储库时显式禁用，例如：

请求：PUT `http://127.0.0.1:9200/_snapshot/s3_repository?verify=false`

参数：

```
{
  "type": "s3",
  "settings": {"bucket": "my_s3_bucket", "region": "eu-west-1"}
}
```

验证过程也可以通过运行以下命令来手动执行：

请求：POST `http://127.0.0.1:9200/_snapshot/s3_repository/_verify`

它返回一个节点列表，成功验证或发送错误消息（如果验证过程失败）。

2. 快照

一个存储库可以包含同一个集群的多个快照。快照在集群内的名称是唯一的。在一个库中创建名称为 `snapshot_1` 快照可以执行下面的命令：

请求：PUT `http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1?wait_for_completion=true`

`wait_for_completion` 参数指定快照初始化后立即返回（默认）还是等待快照完成后返回。在快照初始化时，所有以前的快照信息加载到内存，这意味着创建一个大的库快照可能需要几秒钟（甚至几分钟）。所以有时候 `wait_for_completion` 参数设置为 `false` 也是需要等待一会的。

默认情况下，会创建包含集群中所有打开和启动的索引的一个快照，此行为可以通过指定快照请求中的索引列表来改变，例如：

请求：PUT `http://127.0.0.1:9200/snapshot/my_backup/snapshot_1`

参数:

```
{
  "indices": "index_1,index_2",
  "ignore_unavailable": "true",
  "include_global_state": false
}
```

可以使用支持多索引语法的索引参数来指定快照的索引列表。快照请求也支持 `ignore_unavailable` 选项, 将它设置为 `true` 时可以在快照创建过程中把不存在的索引忽略掉。默认情况下, 当 `ignore_unavailable` 选项不设置或者设置为 `false` 的时候, 如果不存在索引, 索引的快照请求将失败。设置 `include_global_state` 为 `false` 可以防止集群全局状态被存储为快照的一部分。默认情况下, 如果有一个或多个索引没有可用的主分片, 整个快照会失败, 这种行为可以通过设置部分来改变。

索引快照处理是渐进的。在索引快照的过程中 Elasticsearch 会分析索引文件的列表是否已存储在存储库中, 同时复制上次创建的快照或更改的文件。这允许在一个紧凑形式的存储库中保存多个快照, 快照的过程是非阻塞的方式执行。当对索引正在执行快照时, 所有的索引和搜索操作可以继续执行。然而, 快照执行的数据是在快照创建时的时间点确定的, 所以在快照过程开始后添加到索引中的记录将不会在快照中出现。在 1.2.0 版本之前, 如果集群有迁移或者在索引中初始化分片时会失败, 在 1.2.0 之后, 快照的操作会等待这些完成后再操作。一个快照除了创建集群的副本外, 还可以存储全局集群元数据, 其中包括持久性集群设置和模板。临时设置和注册的快照库不会存储为快照的一部分。

在任何时间只有一个快照过程在集群中被执行。而对快照创建分片时, 这分片不能移动到另一个节点, 它可能干扰平衡过程和过滤操作。一次快照操作只能移动一块到另一个节点(根据当前配置过滤设置和调整算法)。一旦一个快照完成, 关于这个快照的信息, 可以使用以下命令来获得:

请求: `GET http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1`

类似的仓库, 对多个快照信息可以通过一次进行查询, 以及支持通配符, 或者用 `_all` 查询所有:

请求: `GET http://127.0.0.1:9200/_snapshot/my_backup/snapshot_*,some_other_snapshot`

如果一些快照不可用, 该命令将失败。布尔参数 `ignore_unavailable` 可用于返回所有目前可用的快照。可以使用以下命令查询当前正在运行的快照:

请求: `GET http://127.0.0.1:9200/_snapshot/my_backup/_current`

快照可以使用下列命令从存储库中删除:

请求: `DELETE http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1`

当快照从库中删除时, Elasticsearch 将删除与快照关联的和其他快照也不使用的所有文件, 如果执行快照创建的过程中同时删除快照, 则快照创建的过程会停止, 同时删除相关的文件。因此, 删除快照操作可以用来取消被错误启动的长时间运行的快照操作。使用下面的

命令可以删除存储库:

请求: DELETE http://127.0.0.1:9200/_snapshot/my_backup

当一个库被删除, Elasticsearch 仅删除存储库的位置的引用, 快照本身并没有被删除。

3. 恢复

使用以下命令可恢复快照:

请求: POST http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1/_restore

默认情况下, 快照中的所有索引以及集群状态将被恢复, 在恢复 include_global_state 选项时, 可以指定具体的索引和集群状态被恢复。索引的列表支持多索引语法。

rename_pattern 和 rename_replacement 选项也可用于重命名索引, 可以使用正则表达式, 该表达式支持引用原始文本。设置 include_aliases 为 false 可以防止相关索引的别名被一起恢复, 例如:

请求: POST http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1/_restore

参数:

```
{
  "indices": "index_1,index_2",
  "ignore_unavailable": "true",
  "include_global_state": false,
  "rename_pattern": "index_(.+)",
  "rename_replacement": "restored_index_$1"
}
```

恢复操作可以在一个功能集群上执行。在快照索引中有相同数量分片的索引如果是关闭的, 则只能恢复。如果索引是关闭的则恢复操作会自动打开索引, 如果索引在集群中不存在, 恢复操作将会创建新的索引, 如果恢复集群状态, 则在该集群中不存在的模板将被添加, 已恢复的模板会替换为具有相同名称的现有模板。恢复中的设置被添加到现有的设置中。

部分恢复, 默认情况下, 如果一个或更多的索引如果没有可用的快照碎片, 整个恢复操作将失败。如果一些碎片没有快照, 通过设置 partial 为 true, 仍可以恢复部分索引。请注意, 在这种情况下只有有分片的索引会被正确恢复, 没有分片的索引会建立空的索引。

在恢复过程中改变索引设置, 在恢复过程中大多数索引的设置可以被覆盖。例如, 下面的命令将不创建任何副本而切换回默认刷新间隔的恢复索引 index_1:

请求: POST http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1/_restore

参数:

```
{
  "indices": "index_1",
  "index_settings": {"index.number_of_replicas": 0},
  "ignore_index_settings": ["index.refresh_interval"]
}
```

请注意, 一些设置, 如 index.number_of_shards 在恢复操作期间是无法更改的。

恢复到不同的集群，快照中存储的信息不依赖于特定的集群或集群名称。因此，它有可能恢复从一个集群到另一个集群的快照。仅要求注册存储库中包含快照，并启动恢复过程，新的集群不具有相同的大小或拓扑结构。如果一个集群有的空间比较小，需要考虑的事情有：首先，要确保新的集群有足够的容量来存储快照中的所有索引。在还原的过程中，可以改变索引设置，以减少复制的次数，这可以帮助将快照恢复到较小的集群，也可以选择使用索引参数。在版本 1.5.0 之前，Elasticsearch 没有检查恢复持续的设置使得有可能恢复出错，参数 `discovery.zen.minimum_master_nodes` 禁用一个小集群直到添加所需的主节点数。从版本 1.5.0 此设置将被忽略。如果在原始集群的索引被分配到特定的节点，使用过滤分片配置，同样的规则将在新的集群执行。因此，如果新的集群不包含具有已有属性的节点，则该恢复的索引可以被分配，除非这些索引分配设置在恢复操作期间改变，否则不会恢复成功。

4. 快照状态

使用以下命令可获得当前运行快照的详细状态信息列表：

请求：GET `http://127.0.0.1:9200/_snapshot/_status`

下面的命令将返回有关当前正在运行的快照的信息。通过指定一个存储库的名称，可以将结果限制为特定的存储库：

请求：GET `http://127.0.0.1:9200/_snapshot/my_backup/_status`

如果指定了存储库名称和快照标识，该命令将返回指定快照的详细状态信息：

请求：GET `http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1/_status`

同样支持多个 id 的查询，例如：

请求：GET `http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1,snapshot_2/_status`

监控快照恢复过程：当快照的进度和恢复正在运行，有几种方法来监视它们。`wait_for_completion` 参数会阻止客户端直到操作完成。这是最简单的方法，可以用来获得有关操作完成的通知。

还可以定期对快照信息进行监视，例如：

请求：GET `http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1`

请注意，快照信息的操作使用相同的资源和线程池作为快照操作。所以，当大的分片进行快照会导致快照操作，在返回结果之前等待可用资源，所以这种情况会占用很多时间。为了获得更直接的快照状态的完整信息，可以使用快照状态命令来查询，例如：

请求：GET `http://127.0.0.1:9200/_snapshot/my_backup/snapshot_1/_status`

快照信息方法在处理过程中返回快照的基本信息，快照状态返回在快照中每个分片的当前状态。

在恢复操作时，Elasticsearch 的集群通常会进入红色状态。这是因为还原操作开始恢复索引的主要碎片。在这个操作时，主分片变得不可用，这体现在红色状态。一旦碎片恢复完成时，Elasticsearch 转换为标准的复制过程，创建所需数量的副本，这时集群切换到黄色状

态。一旦所有所需的复制品被创建，集群切换到绿色状态。集群健康操作仅为恢复过程的高层次状态。通过使用索引恢复和精简 API 可以获得更详细的恢复过程中的状态。

停止当前正在运行的快照和恢复操作：快照和恢复操作允许一次只运行一个快照或一次恢复。如果一个当前运行的快照被错误地执行，或者需要很长的时间，可以使用快照删除操作来终止它。快照删除时会进行检查，如果删除快照正在运行，则停止此次删除。

还原操作使用标准的分片恢复机制。因此，可以通过删除索引来取消恢复的操作。请注意，在这种情况下所有已删除索引的数据将被从集群中移除。

快照和恢复操作都受集群和索引块的影响。例如，注册和注销库需要通过编写全局元数据来进行访问。快照操作需要所有的索引和元数据以及全局元数据都是可读的。恢复操作需要全局元数据是可写的，但是索引水平块将被忽略，因为在恢复的过程中索引是重建的。请注意，存储库内容不是集群的一部分，因此集群块不影响内部存储库操作，例如已注册的存储库中的列表或删除快照。

8.4 线程池配置

节点包含多个线程池用来改善节点的线程内存消耗。这些线程池有许多与之关联的队列，用于保留等待的请求而不是丢弃。

有多种线程池，最主要的见表 8-7。


表 8-7 主要线程池

名称	说明
generic	用于通用操作（例如，背景节点发现）。线程池类型是 cached
index	用于索引 / 删除操作。线程池类型是 fixed，大小是 # of available processors，队列大小是 200
search	用于统计 / 搜索操作。线程池类型是 fixed，大小是 $\text{int}((\# \text{ of available_processors} * 3) / 2) + 1$ ，队列大小是 1000
suggest	用于建议操作。线程池类型是 fixed，大小是 # of available processors，队列大小是 1000
get	用于获取操作。线程池类型是 fixed，大小是 # of available processors，队列大小是 1000
bulk	用于大量操作。线程池类型是 fixed，大小是 # of available processors，队列大小是 50
percolate	用于渗出操作。线程池类型是 fixed，大小是 # of available processors，队列大小是 1000
snapshot	用于快照 / 恢复操作。线程池类型是 scaling 保持时间是 5m，大小是 $\text{min}(5, (\# \text{ of available processors})/2)$
warmer	用于索引预加载操作。线程池类型是 scaling 保持时间是 5m，大小是 $\text{min}(5, (\# \text{ of available processors})/2)$
refresh	用于刷新操作。线程池类型是 scaling 保持时间是 5m，大小是 $\text{min}(10, (\# \text{ of available processors})/2)$
listener	主要用于 Java 客户端执行时，监听线程设置为 true。线程池类型是 scaling 默认大小是 $\text{min}(10, (\# \text{ of available processors})/2)$

更改特定的线程池可以通过设置指定类型参数。例如，修改 index 线程池拥有更多的线

程如下所示：

```
threadpool:
  index:
    size: 30
```

 **注意** 可以使用集群更新设置来动态升级线程池设置。

1. 线程池类型

下面是线程池类型和它们各自的参数：

- ❑ **cached**。cached 线程池是一个无限的线程池，如果存在挂起的请求时，就会产生一个线程。这个线程池用来防止提交的请求被阻塞或丢弃。线程池中的无用线程会在持续期满（默认为 5 分钟）之后被销毁。cached 线程池被保留为通用线程池。keep_alive 参数决定一个线程的空闲时间：

```
threadpool:
  generic:
    keep_alive: 2m
```

- ❑ **fixed**。fixed 线程池拥有固定大小的线程来操作队列中的请求（任意界限）直到请求没有线程提供服务。size 参数控制线程的数量。queue_size 参数可以控制没有线程执行的请求队列的大小。默认设置为 -1，意味着无限大。当请求到达而且队列已经满了，请求会被中止，如下所示：

```
threadpool:
  index:
    size: 30
    queue_size: 1000
```

- ❑ **scaling**。scaling 线程池拥有动态数量的线程。线程的数量与工作量成正比，并且在 1 和 size 参数值之间变化。keep_alive 参数决定一个线程的空闲时间：

```
threadpool:
  warmer:
    size: 8
    keep_alive: 2m
```

2. 处理器设置

处理器的数量是自动检测的，线程池的设置会基于结果自动设置。有时，处理器的数量会被错误检测，在这种情况下，处理器的数量可以使用 processors 进行明确设置。

为了检查处理器检测的数量，可以使用节点信息，节点利用 os 标示。

8.5 索引配置

索引模块是控制每个索引指标的模块。索引模块包括分词、分片控制和分配、字段映射、索引相似性配置、慢查询记录、文件系统配置、控制事务和刷新模块。

相似性定义了如何对文件进行评分机制，相似性对应的维度是列，这意味着可以对每一列进行不同相似性的配置。Elasticsearch 允许用户不作任何配置使用 default 和 BM25 相似度模型，因为它们是预先在系统中配置好的。如果是 DFR 和 IB 模型，我们需要配置才能使用。

索引分片配置主要有：分片配置过滤（Shard allocation filtering）、延迟分配（Delayed allocation）和分片总数配置（Total shards per node）。

8.5.1 缓存配置

1. 总内存控制

在 Elasticsearch 中有很多控制器可以防止内存溢出，每个控制器可以指定内存使用的最大值，除此之外，还有一个总的控制器在确定整个系统使用的最大内存值。这些配置都可以动态更新。总的内存控制有以下参数：

❑ `indices.breaker.total.limit`：总的内存使用大小，默认为 JVM 堆内存大小的 70%。

2. 列数据内存控制

列数据内存控制器在 Elasticsearch 在加载数据前系统会估计有多少数据，如果估计超过设定的阈值，就会产生一个异常来阻止该字段的数据加载。

❑ `indices.breaker fielddata.limit`：列数据内存大小的限制，默认为 JVM 堆内存大小的 60%。

❑ `indices.breaker fielddata.overhead`：所有列估计的内存大小的乘积，默认是 1.03。

3. 请求内存控制

请求控制器是防止 Elasticsearch 每个请求的数据结构超过阈值：

❑ `indices.breaker.request.limit`：请求控制器的大小，默认为 JVM 堆内存大小的 40%。

❑ `indices.breaker.request.overhead`：所有请求的乘积，默认值为 1。

4. 数据缓存

现场数据缓存主要用于排序或聚合操作时，它将所有的字段值加载到内存中以便快速访问文档中的这些值。其中有一个参数：

❑ `indices.fielddata.cache.size`：数据缓存的最大值，可以是一个节点的堆内存大小的比例，例如 30%，也可以是一个绝对数字，比如 12GB。默认是无限限制，可以最大的利用内存。这个配置是静态的配置，必须在集群中的每个数据节点启动前配置好。可以通过 `http://127.0.0.1:9200/_nodes/stats` 请求来监控节点的使用情况。

5. 节点查询缓存

查询缓存负责缓存查询的结果。每个节点都有一个查询缓存，缓存为这个节点下的所有分片服务。缓存采用最近最少使用算法。当缓存满时，把最少使用的数据优先删掉。查询缓存只有使用过滤的时候才会起作用。其中有一个参数：

- ❑ `indices.queries.cache.size`：可以是一个节点的堆内存大小的比例，例如 5%，也可以是一个绝对数字，比如 512MB。默认为 JVM 堆内存大小的 10%。


6. 索引缓冲区

索引缓冲区用于存储新的索引文档。当缓冲区满后，缓冲区中的文件写入磁盘上的一个段，会在节点的所有分片上分离。它的设置是静态的，并且必须在集群中的每个数据节点上配置：

- ❑ `indices.memory.index_buffer_size`：一个节点索引缓冲区的大小，可以是一个节点的堆内存大小的比例或者是一个绝对数字。默认为 JVM 堆内存大小的 10%。
- ❑ `indices.memory.min_index_buffer_size`：可以使用此设置指定最小的索引缓冲区大小。默认值为 48MB。
- ❑ `indices.memory.max_index_buffer_size`：可以使用此设置指定最大的索引缓冲区大小。默认为无限制。
- ❑ `indices.memory.min_shard_index_buffer_size`：设置分配给每个分片索引缓冲区的内存最小值，默认值为 4MB。

7. 分片请求缓存

当一个搜索请求是对一个索引或者多个索引的时候，每一个分片都是进行它自己内容的搜索，然后把结果返回到协调节点，然后把这些结果合并到一起统一对外提供。分片缓存模块缓存了这个分片的搜索结果。这使得搜索频率高的请求会立即返回。

 **注意** 请求缓存只缓存查询条件 `size=0` 的搜索，缓存的内容有 `hits.total`、`aggregations`、`suggestions`，不缓存原始的 `hits`。通过 `now` 查询的结果将不缓存。

缓存失效：只有在分片的数据实际上发生了变化时刷新分片缓存才会失效。刷新的时间间隔越长，缓存的数据越多，当缓存不够的时候，最少使用的数据将被删除。缓存过期可以手工设置，例如：

```
127.0.0.1:9200/kimchy,elasticsearch/_cache/clear?request_cache=true
```

默认情况下缓存未启用，但在创建新的索引时可启用，例如：

请求：PUT <http://127.0.0.1:9200/secisland>

```
{
  "settings": {"index.requests.cache.enable": true}
```



```
}
```

当然也可以通过动态参数配置来进行设置：

请求：PUT http://127.0.0.1:9200/secisland/_settings -d'

```
{ "index.requests.cache.enable": true }
```

每请求启用缓存，查询字符串参数 `request_cache` 可用于启用或禁用每个请求的缓存。

例如：

请求：PUT http://127.0.0.1:9200/secisland/_search?request_cache=true

```
{
  "size": 0,
  "aggs": {
    "popular_colors": {
      "terms": { "field": "colors" }
    }
  }
}
```



注意 如果你的查询使用了一个脚本，其结果是不确定的（例如，它使用一个随机函数或引用当前时间）应该设置 `request_cache` 为 `false` 禁用请求缓存。

数据缓存是根据 `key` 来确定是否为相同内容，`key` 的值是整个 JSON，这意味着如果 JSON 发生了变化，例如如果输出的顺序不同，缓存的内容将会不同。不过大多数 JSON 库对 JSON 键的存放顺序是固定的。

分片请求缓存是在节点级别进行管理的，并有一个默认的值是 JVM 堆内存大小的 1%，可以通过配置文件进行修改。

例如：`indices.requests.cache.size: 2%`

可以通过 `127.0.0.1:9200/_stats/request_cache?pretty&human` 或者 `'127.0.0.1:9200/_nodes/stats/indices/request_cache?pretty&human` 来缓存监控，缓存的大小（以字节为单位）。

8. 索引恢复

下面的设置是索引恢复配置，索引恢复的内容详见 2.5.3 节。

`indices.recovery.concurrent_streams`：默认值为 3。

`indices.recovery.concurrent_small_file_streams`：默认值为 2。

`indices.recovery.file_chunk_size`：默认值为 512KB。

`indices.recovery.translog_ops`：默认值为 1000。

`indices.recovery.translog_size`：默认值为 512KB。

`indices.recovery.compress`：默认值为 `true`。

`indices.recovery.max_bytes_per_sec`：默认值为 40MB。

9. TTL 区间

文档有个 ttl 值，可以设置当过期的时候是否需要删除，参数如下：

- `indices.ttl.interval`：删除程序的运行时间。默认为 60s。
- `indices.ttl.bulk_size`：删除处理与批量请求的数量，默认为 10000。

8.5.2 索引碎片分配

这个模块提供每个索引设置来控制碎片分配到节点。

1. 碎片分配过滤

控制哪个碎片被分配到哪个节点。在启动时，可能随机将元数据属性分配给各个节点。

例如，节点可能被分配 rack 或 group 属性：

```
bin/elasticsearch --node.rack rack1 --node.size big
```

这些属性设置也可以在 `elasticsearch.yml` 设置文件中进行配置。

元数据属性可以使用 `index.routing.allocation.*` 设置来分配索引到特定的节点组。例如将 test 索引移动到 big 或 medium 节点：

请求：PUT `http://127.0.0.1:9200/test/_settings`

```
{
  "index.routing.allocation.include.size": "big,medium"
}
```

或者，从 small 节点中移除 test 索引：

请求：PUT `http://127.0.0.1:9200/test/_settings`

```
{"index.routing.allocation.exclude.size": "small"}
```

可以指定多个规则，在这种情况下，所有条件必须满足：

请求：PUT `http://127.0.0.1:9200/test/_settings`

```
{
  "index.routing.allocation.include.size": "big",
  "index.routing.allocation.include.rack": "rack1"
}
```

下面的设置是动态的，允许活动索引从一个节点移动到另一个节点：

- `index.routing.allocation.include.{attribute}`：分配索引到一个节点，节点的 {attribute} 拥有至少一个逗号分隔的值。
- `index.routing.allocation.require.{attribute}`：分配索引到一个节点，节点的 {attribute} 拥有所有逗号分隔的值。
- `index.routing.allocation.exclude.{attribute}`：分配索引到一个节点，节点的 {attribute} 不拥有任何逗号分隔的值。

也支持这些特殊的属性：

□ `_name`: 通过节点名匹配节点。

□ `_host_ip`: 通过主机 IP 地址匹配节点 (IP 和主机名)。

□ `_publish_ip`: 通过发布 IP 地址匹配节点。

□ `_ip`: 通过 `_host_ip` 或 `_publish_ip` 进行匹配。

□ `_host`: 通过主机名匹配节点。

所有属性值可以用通配符指定:

请求: `PUT http://127.0.0.1:9200/test/_settings`

```
{
  "index.routing.allocation.include_ip": "192.168.2.*"
}
```

2. 延迟分配

延迟分配由于节点离开造成的未赋值碎片。

当节点离开集群时, 主要的反应是:

□ 提升副本分片为主分片来替换节点中的任何主分片。

□ 分配副本分片替换缺失副本 (假设有足够的节点)。

□ 分片再平衡, 均匀地分配到其余节点。

这些行为的目的是保护集群防止数据丢失, 通过确保每个分片尽可能快地完全复制。

即使我们在节点级别和集群级别同时改善节流, “分片移动” 仍然会给集群带来许多不必要的额外负载, 如果缺失的节点可能很快地返回集群。想象如下情景:

□ 节点 5 丢失网络连接。

□ 主节点提升节点 5 中的所有主分片的副本分片成为主分片。

□ 主节点分配新的副本到集群中的其他节点。

□ 每个新副本通过网络获取主分片的整体复制。

□ 更多的分片被移动到不同的节点使集群再平衡。

□ 几分钟后, 节点 5 重新连接到集群中。

□ 主节点通过分配分片到节点 5 使集群再平衡。

如果主节点只等了几分钟, 然后缺失分片可以使用最小的网络流量重新分配给节点 5。这个过程对于被动态同步刷新的空分片会更快速。

副本分片的分配可以通过动态设置 `index.unassigned.node_left.delayed_timeout` 延迟, 默认为 1m。

可以在一个活动索引 (或所有索引) 上更新设置:

请求: `PUT http://127.0.0.1:9200/_all/_settings`

```
{
  "settings": { "index.unassigned.node_left.delayed_timeout": "5m" }
}
```

启用延时分配，上面的情况会变成这样：

- ❑ 节点 5 丢失网络连接。
- ❑ 主节点提升节点 5 中的所有主分片的副本分片成为主分片。
- ❑ 主节点记录未赋值节点分配的延时信息。
- ❑ 集群状态仍然是黄色，因为有未赋值的副本分片。
- ❑ 几分钟后，在延时超时之前，节点 5 返回集群。
- ❑ 缺失副本被重分配给节点 5（同步刷新的分片几乎立即恢复）。

（1）取消分片移动

如果延时分配超时，主节点会开始恢复缺失分片到另一个节点。如果缺失节点重新加入集群中，并且它的主分片仍然有相同的同步 ID，分片移动会被取消并且同步分片会用于恢复。

由于这个原因，默认的 timeout 设置为一分钟：即使分片开始移动，取消恢复同步分片的花销也是小的。

（2）监控未赋值分片的延时

分配被延时的分片数量可以通过集群健康接口查看：

请求：GET http://127.0.0.1:9200/_cluster/health

（3）永久移除节点

如果一个节点不会返回集群并且希望 Elasticsearch 立即分配缺失分片，只需要设置超时为零：

请求：PUT http://127.0.0.1:9200/_all/_settings

```
{
  "settings": {"index.unassigned.node_left.delayed_timeout": "0"}
}
```

可以在缺失分片已经开始恢复的时候重设延时。

3. 每个节点的总碎片

每个节点对相同的索引限制碎片数量。集群级别的分片分配尝试传播单个索引的分片到尽可能多的节点中。然而，取决于拥有的分片和索引数量，以及它们的大小，可能并不总是可以均匀地传播分片。

下面的动态设置可以强制限制每个节点允许单个索引分片的总数量：

- ❑ `index.routing.allocation.total_shards_per_node`：分配到单个节点的最大分片数量（副本和主分片）。默认没有限制。

可以限制节点的分片数量，不管索引的数量：

- ❑ `cluster.routing.allocation.total_shards_per_node`：分配到单个节点的最大总分片数量（副本和主分片）。默认没有限制。

8.5.3 合并

一个 Elasticsearch 分片就是一个 Lucene 索引，Lucene 索引被分解为分片。分片是索引的内部存储单元，存储索引数据并且是不变的。周期性合并（merge）小的分片为更大的分片来保持索引大小在范围内。

合并执行使用自动节流来平衡合并和其他活动（比如搜索）之间的硬件资源的使用。

合并计划（同时合并计划）控制合并操作的执行。合并单独的线程执行，当线程达到最大值，更多的合并会等待直到合并线程可用。

合并计划支持下列动态设置：`index.merge.scheduler.max_thread_count` 指一次合并的最大线程数。默认值是 `Math.max(1, Math.min(4, Runtime.getRuntime().availableProcessors()/2))` 对于固态硬盘（SSD），有很好的效果。如果索引在机械硬盘上，默认值减少到 1。

8.5.4 相似模块

相似性（得分 / 排名模型）定义了匹配文档如何进行评分。相似性是针对字段的，意味着通过映射可以对每个字段定义不同的相似性模块。

1. 配置相似性

大多数已经存在或自定义的相似性拥有配置选项，可以通过索引设置进行配置。当创建索引或更新索引设置的时候，可以提供索引选项。

```
"similarity" : {  
  "my_similarity" : { "type" : "DFR", "basic_model" : "g",  
    "after_effect" : "l1", "normalization" : "h2", "normalization.h2.c" : "3.0" }  
}
```

我们可以在映射中引用自定义相似性模块：

```
{  
  "book" : {  
    "properties" : { "title" : { "type" : "string", "similarity" : "my_similarity" } }  
}
```

2. 可用的相似性模块

默认相似性模块基于 TF/IDF 模式。拥有选项如下：

`discount_overlaps`：决定重叠词元（词元的位置增量为 0）是否要被忽略。默认为 true，意味着重叠词元不会被统计。

类型名：default。

BM25 相似性模块：基于 TF/IDF 的相似性模块拥有内置的 tf 标准并且对短字段（比如名字）效率更高。拥有下列参数：

□ `k1`：控制非线性索引词频率标准（饱和度）。

□ `b`：控制文档长度标准化到 tf 值的程度。

❑ `discount_overlaps`：决定重叠词元（词元的位置增量为 0）是否要被忽略。默认为 `true`，意味着重叠词元不会被统计。

类型名：BM25

DFR 相似性模块：实现随机性框架的分支。拥有下列参数：

❑ `basic_model`：可能的值有：`be`、`d`、`g`、`if`、`in`、`ine` 和 `p`。

❑ `after_effect`：可能的值有：`no`、`b` 和 `l`。

❑ `normalization`：可能的值有：`no`、`h1`、`h2`、`h3` 和 `z`。

类型名：DFR

IB 相似性模块：基于信息的模式。算法基于设想，信息内容是通过基本元素的重复使用产生的符号分布序列。对于书面文本，这个方式会对比不同作者的写作风格。拥有下面的选项：

❑ `distribution`：可能的值：`ll` 和 `spl`。

❑ `lambda`：可能的值：`df` 和 `tff`。

❑ `normalization`：和 DFR 相似模块相同。

类型名：IB

LM Dirichlet 相似性模块：拥有这些选项：

`mu`：默认为 2000。

类型名：LMDirichlet。

LM Jelinek Mercer 相似性模块

这个算法视图捕捉文本中的重要样品，拥有下面选项：

`lambda`：最佳值取决于采集和查询。标题查询的最佳值大约是 0.1，长查询的最佳值是 0.7。当值接近于 0，匹配更多查询索引词的文档会比匹配较少索引词的文档的排列位置更靠前。

类型名：LMJelinekMercer

默认和基础相似性模块

默认情况下，Elasticsearch 会使用任何配置为 `default` 的相似性模块。然而，相似性方法 `queryNorm()` 和 `coord()` 不是每个字段都会执行。因此，对于专家用户想要改变这两种方法的实现，不会更改 `default`，可以用 `base` 名来配置相似性。然后，相似性会用于这两种方法。

通过在 `elasticsearch.yml` 中进行配置可以对所有字段修改默认相似性模块：

```
index.similarity.default.type: BM25
```

8.5.5 响应慢日志监控

在执行 Elasticsearch 操作的时候，有的操作会占用大量的资源导致响应很慢，这个时候就需要对 Elasticsearch 的操作进行监控，找到那些响应很慢的请求。Elasticsearch 的请求主要分为搜索和索引，Elasticsearch 分别提供了这两种类型的监控配置。

1. 搜索慢日志

慢搜索日志配置可以记录响应慢的搜索（查询和获取阶段）并将其放到一个专门的日志文件，这个配置只针对当前分片节点有效。

可以对执行的查询阶段和获取阶段的阈值进行分别配置，具体配置在 `elasticsearch.yml` 文件中：

```
index.search.slowlog.threshold.query.warn: 10s
index.search.slowlog.threshold.query.info: 5s
index.search.slowlog.threshold.query.debug: 2s
index.search.slowlog.threshold.query.trace: 500ms

index.search.slowlog.threshold.fetch.warn: 1s
index.search.slowlog.threshold.fetch.info: 800ms
index.search.slowlog.threshold.fetch.debug: 500ms
index.search.slowlog.threshold.fetch.trace: 200ms
```

这些设置是动态的，而且可以对每个索引进行设置。

默认情况下，搜索慢日志没有被启用（设置为-1）。日志级别（warn、info、debug、trace）可以通过给日志分级来控制日志的记录。不是所有级别的日志都需要记录，多个级别的好处是可以根据级别减少日志的数量，根据业务需要只关注重点的日志。

日志记录是在分片级别范围完成的，意味着在特定分片中执行查询请求，不包含整个可以广播到多个分片执行的搜索请求。与请求级别相比，分片级别日志记录的好处是关联特定机器上的实际执行操作。

默认的配置文件的在 `logging.yml` 文件中，默认的配置如下：

```
index_search_slow_log_file:
  type: dailyRollingFile// 日志类型，每天一个文件
  file: ${path.logs}/${cluster.name}_index_search_slowlog.log// 文件命名格式
  datePattern: "'.'yyyy-MM-dd"// 每日备份的后缀
  layout:
    type: pattern
    conversionPattern: "[%d{ISO8601}][%-5p][%-25c] %m%n"// 记录日志的开头格式
```

2. 索引慢日志

索引慢日志类似于搜索慢日志的功能。日志文件以 `_index_indexing_slowlog.log` 结尾。日志和阈值可以在 `elasticsearch.yml` 文件中配置：

```
index.indexing.slowlog.threshold.index.warn: 10s
index.indexing.slowlog.threshold.index.info: 5s
index.indexing.slowlog.threshold.index.debug: 2s
index.indexing.slowlog.threshold.index.trace: 500ms
index.indexing.slowlog.level: info
index.indexing.slowlog.source: 1000
```

默认情况下，Elasticsearch 会记录 `_source` 中的前 1000 个字符到慢日志中。可以用 `index.indexing.slowlog.source` 进行修改。设置为 `false` 或 `0` 会完全跳过日志记录源，设置为

true 会记录整个源（无论有多大）。

默认的配置文件是在 logging.yml 文件中，默认的配置如下，具体的说明同搜索慢日志：

```
index_indexing_slow_log_file:
  type: dailyRollingFile
  file: ${path.logs}/${cluster.name}_index_indexing_slowlog.log
  datePattern: "'.'yyyy-MM-dd"
  layout:
    type: pattern
    conversionPattern: "[%d{ISO8601}][%-5p][%-25c] %m%n"
```

8.5.6 存储

存储模块可以控制索引数据如何在磁盘上存储和访问。

有不同文件系统的实现或存储类型。会自动选择最佳的操作环境：Windows 64 位系统（mmapfs）、Windows 32 位系统（simplefs）以及其他系统（default，混合 niofs 和 mmapfs）。

可以通过添加 config/elasticsearch.yml 文件配置对所有索引进行优化：

```
index.store.type: niofs
```

可以在每个索引创建时对其进行静态设置：

```
PUT /secisland
{
  "settings": {"index.store.type": "niofs"}
}
```

支持的存储类型包括：

- ❑ simplefs——简单文件系统类型是文件存储系统的简单实现（对应 Lucene 的 SimpleFsDirectory），使用随机存取文件。这一实现的并发性能比较差，多线程会受到阻塞。当需要保留索引的时候，通常会使用 niofs
- ❑ niofs——NIO 文件系统类型在文件系统上使用 NIO 存储分片索引（对应 Lucene 中的 NIOFSDirectory）。多线程可以从同一个文件同时读取数据。Windows 系统中不推荐使用这个类型，因为 SUN Java 的实现存在漏洞
- ❑ mmapfs——MMAP 文件系统类型在文件系统上通过映射文件到内存（mmap）存储分片索引（对应 Lucene 中的 MMapDirectory）。内存映射在进程中使用了虚拟内存地址空间的一部分，空间大小等于映射文件的大小。使用这个类型之前，要确保拥有充足的虚拟地址空间
- ❑ default_fs——默认类型是 NIO FS 和 MMapFS 的混合，对每个类型的文件选择最佳文件系统。当前只有 Lucene 索引词字典和文件映射到内存的文档用来减少对操作系统的影响。所有其他文件使用 Lucene 的 NIOFSDirectory 打开。如果索引词字典特别大，可以应用地址空间设置（例如，sysctl -w vm.max_map_count=262144，或者在 /etc/sysctl.conf 文件中修改 vm.max_map_count 进行永久设置）

8.5.7 事务日志

Lucene 的修改只有在提交的时候保存在磁盘上，这是个相对沉重的操作，所以不能在每次索引或删除操作之后执行。一次提交之后另一次提交之前如果发生了进程退出或硬件故障的事件，这之间的修改会丢失。

为了防止数据丢失，每个分片都有事务日志（translog）或与之相关的日志。任何索引或删除操作在 Lucene 索引内部执行之后会被写入事务日志。

如果发生了崩溃事件，当分片恢复的时候，最近的事务可以根据事务日志进行恢复。

Elasticsearch 冲洗（flush）操作是执行 Lucene 提交并开始新的事务日志的过程。会自动进行后台执行来确保事务日志不会变得太大，这会使重做操作在恢复时花费大量的时间。可以通过接口调用，虽然很少需要手动执行。

1. 冲洗设置

下面的动态更新设置控制内存缓冲区的数据保存到磁盘的频率：

- ❑ `index.translog.flush_threshold_size`——一旦事务日志达到了这个大小，冲洗操作就会执行，默认值为 512MB
- ❑ `index.translog.flush_threshold_ops`——多少个操作之后执行冲洗。默认是 unlimited
- ❑ `index.translog.flush_threshold_period`——无论事务日志的大小是多少，触发冲洗操作需要等待的时间，默认值为 30m
- ❑ `index.translog.interval`——多久检查一次是否需要执行冲洗操作。在间隔值和 2 倍间隔值之间的随机值。默认值为 5s

2. 事务日志设置

事务日志的数据仅在事务同步和提交之后保存在磁盘上。如果发生了硬件错误，前一次事务日志提交之后的任何数据写入都会丢失。

默认情况下，Elasticsearch 在任何索引、删除、更新或大量请求之后会提交事务日志。事实上，在事务日志被成功地同步和提交给主分片和任何分配的副本分片之后，Elasticsearch 只会报告成功执行的请求给客户。

每个索引使用下面动态更新的设置控制事务日志的行为：

- ❑ `index.translog.sync_interval`——事务日志多久被提交和同步到磁盘上。默认值为 5s
- ❑ `index.translog.durability`——请求完成之后，事务日志是否同步和提交。接收这些参数：
 - `request`（默认）每次请求之后执行同步和提交。在硬件失败的情况下，所有收到的写入会事先提交到磁盘上
 - `async` 经过一个同步间隔，都会后台执行同步和提交。在硬件失败的情况下，自从最后一次自动提交之后收到的写入都会被丢弃
- ❑ `index.translog.fs.type`——是否在内存中缓存事务日志。接收这些参数：
 - `buffered`（默认）事务日志首先写入内存中 64KB 大小的缓存，并且只有当缓存填

满的时候才会写入磁盘，或者，同步被写请求或同步间隔触发

- simple 事务日志会立即写入文件系统，不用缓存。然而，这些写入只会在同步和提交被触发时才会保存在磁盘上

8.6 小结

本章介绍了 Elasticsearch 的高级设置功能，利用这些配置可以管理集群与扩展，并且在 Elasticsearch 中可以创建快照，利用快照可以进行集群的整体恢复或部分恢复。

9.1.2 结构

Watchers 的配置主要包括 Trigger、Input、Condition、Action 几个部分。在执行 Action 之前，可以定义 Transform 以其中处理管道中的内容。Watcher 是弹性搜索集群中的一个组件，它负责监控集群中的各种指标，并在满足特定条件时触发相应的操作。它可以通过 REST API 进行配置和管理，也可以通过 Elasticsearch 的配置文件进行配置。

参数	描述
Trigger	定义何时触发告警，可以是基于时间、基于指标或基于事件。
Input	指定从哪个源获取数据，可以是 Elasticsearch 的索引、日志或外部数据源。
Condition	定义触发告警的条件，可以是基于指标、基于事件或基于自定义逻辑。
Transform	定义在触发告警之前对数据进行处理的步骤，可以是基于 Elasticsearch 的 DSL 或基于自定义脚本。
Actions	定义在触发告警后要执行的操作，可以是发送邮件、发送短信或调用外部 API。

1. Trigger
每个告警必须有一个触发器 (Trigger)，用来定义何时开始检查条件。触发器可以是基于时间的、基于指标的或基于事件的。基于时间的触发器可以按固定的时间间隔触发，也可以按特定的时间点触发。基于指标的触发器可以在满足特定条件时触发，例如当某个指标的数值超过某个阈值时。基于事件的触发器可以在发生特定事件时触发，例如当某个索引的大小发生变化时。

触发器用系统时间来定义当前时间，所以要确保正确的触发时间，需要同步所有节点上的时钟。系统提供以下四种类型的触发器：
• cron (cron 表达式)
• interval (固定间隔)
• hourly (每小时)
• lyhourly (每小时)

告警、监控和权限管理

本章介绍 Elasticsearch 官方支持的几个比较好的插件：Watcher、Marvel、Shield，用这些插件可以对 Elasticsearch 进行告警监控、认证权限管理，不过这几个插件都是收费的。

9.1 告警

Watcher 是进行告警和通知的插件，可以根据数据的变化采取行动。它的设计原理是在 Elasticsearch 中执行查询，满足条件的情况下，产生告警。简单地定义查询、设置限定条件、设置预定计划和将要进行的操作后，Watcher 会自动完成剩下的操作。

9.1.1 安装

Watcher 是以 Elasticsearch 插件的形式存在的，所以安装的过程和插件安装基本一致。安装步骤如下。

1) 在 Elasticsearch 中安装 Watcher 插件：

```
bin/plugin install license
bin/plugin install watcher
```

2) 权限设置，在安装 Watcher 的时候会提示额外权限说明如下：

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@      WARNING: plugin requires additional permissions      @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
* java.lang.RuntimePermission getClassLoader
* java.lang.RuntimePermission setContextClassLoader
* java.lang.RuntimePermission setFactory
```


See <http://docs.oracle.com/javase/8/docs/technotes/guides/security/permissions.html>
for descriptions of what these permissions allow and the associated risks.

Continue with installation? [y/N]y

3) 如果在 Elasticsearch 中设置了禁止自动创建索引的设置, 需要在配置文件中添加如下设置: `action.auto_create_index: .watches,.triggered_watches,.watcher-history*`。

4) 启动 Elasticsearch:

`bin/elasticsearch`

5) 验证是否安装成功:

请求: `GET http://localhost:9200/_watcher/stats?pretty`

返回值:

```
{
  "watcher_state": "started",
  "watch_count": 0,
  "execution_thread_pool": {"queue_size": 0, "max_size": 0}
}
```

9.1.2 结构

Watchex 的配置主要包括 Trigger、Input、Condition、Action 几个部分, 参见表 9-1。此外, 在执行 Action 之前, 可以定义 Transform 过程中处理告警中的内容。

表 9-1 结构说明

参数	描述
Trigger	确定什么时候告警进行检查, 一个告警中至少有一个 Trigger
Input	将数据加载到告警中的内容。如果没有指定输入, 则加载一个空的内容
Condition	控制是否执行告警操作。如果没有指定条件, 条件默认总是执行
Transform	您可以定义在监视级别上的转换, 也可以定义特定于动作的转换, 可选条件
Actions	指定当告警条件被满足时会发生的动作

1. Trigger

每个告警必须有一个触发器 (Trigger), 用来定义开始执行的时间点。每当创建一个触发器时, 将这个触发器注册到一个合适的触发器引擎中。所有的触发器都是基于时间的类型。

触发器用系统时钟来确定当前的时间。所以要确保正确的触发预期时, 需要同步所有节点上的时钟。系统提供以下几种类型的触发器: hourly、daily、weekly、monthly、yearly、cron、interval。

(1) hourly (小时时刻表)

小时时刻表可以指定每小时执行一个计划, 可以设置每小时的几分钟开始执行, 如果不

设置分钟，默认是每小时的 0 分开始执行。

例如，以下每小时时间表触发器在每小时的第 30 分钟执行，12:30, 13:30, 14:30, 等等。

```
{
  ...
  "trigger" : {
    "schedule" : { "hourly" : { "minute" : 30 } }
  }
  ...
}
```

在小时时刻表中可以设置多个分钟执行，例如：

```
{
  ...
  "trigger" : { "schedule" : { "hourly" : { "minute" : [ 0, 15, 30, 45 ] } } }
  ...
}
```

(2) daily (天时刻表)

天时刻表指定每一天在某一特定时间执行一个计划。要配置某天的时间表，需单独指定时间与属性。如果不指定，默认是 0 点执行。例如，下面的作息时间表触发一次每天下午 5:00 执行。

```
{
  ...
  "trigger" : {
    "schedule" : { "daily" : { "at" : "17:00" } }
  }
  ...
}
```

在天时刻表中可以设置多个时间点执行，例如下面的设置在每天的 00:00、12:00 和 17:00 执行：

```
{
  ...
  "trigger" : {
    "schedule" : { "daily" : { "at" : [ "midnight", "noon", "17:00" ] } }
  }
  ...
}
```

midnight 和 noon 是保留字，表示 00:00 和 12:00。

(3) weekly (周时刻表)

周时刻表指定每一周在某一特定时间执行一个计划。可以指定一周的名称、缩写或数字（星期日是第一天的）：

sunday, monday, tuesday, wednesday, thursday, friday and saturday
 sun, mon, tue, wed, thu, fri and sat
 1, 2, 3, 4, 5, 6 and 7

例如下面的设置表示每周五下午 5 点执行:

```
{
  ...
  "trigger" : {
    "schedule" : { "weekly" : { "on" : "friday", "at" : "17:00" } }
  }
  ...
}
```

每周的时刻表也可以设置多个时间点执行, 例如:

```
{
  ...
  "trigger" : {
    "schedule" : {
      "weekly" : [
        { "on" : "tuesday", "at" : "noon" },
        { "on" : "friday", "at" : "17:00" }
      ]
    }
  }
  ...
}
```

或者:

```
{
  ...
  "trigger" : {
    "schedule" : {
      "weekly" : {
        "on" : [ "tuesday", "friday" ],
        "at" : [ "noon", "17:00" ]
      }
    }
  }
  ...
}
```

(4) monthly (月时刻表)

月时刻表指定每一月在某一特定时间执行一个计划。

例如下面的设置表示每月 10 日中午 12 点执行:

```
{
  ...
  "trigger" : {
    "schedule" : {
      "monthly" : { "on" : "10", "at" : "noon" }
    }
  }
  ...
}
```

```

    "monthly" : { "on" : 10, "at" : "noon" }
  }
  ...
}

```

每月的时刻表也可以设置多个时间点执行, 例如:

```

{
  ...
  "trigger" : {
    "schedule" : {
      "monthly" : [
        { "on" : 10, "at" : "noon" },
        { "on" : 20, "at" : "17:00" }
      ]
    }
  }
  ...
}

```

或者:

```

{
  ...
  "trigger" : {
    "schedule" : {
      "monthly" : {
        "on" : [ 10, 20 ],
        "at" : [ "midnight", "noon" ]
      }
    }
  }
  ...
}

```

(5) yearly (年时刻表)

年时刻表指定每一年在某一特定时间执行一个计划。可以指定月的名称、缩写或数字:

```

january, february, march, april, may, june, july, august, september, october,
november, december
jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec
1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

```

例如下面的设置表示每年1月10日中午12点执行:

```

{
  ...
  "trigger" : {
    "schedule" : { "yearly" : { "in" : "january", "on" : 10, "at" : "noon" } }
  }
}

```

每年的时刻表也可以设置多个时间点执行，例如：

```
{
  ...
  "trigger" : {
    "schedule" : {
      "yearly" : [
        { "in" : "january", "on" : 10, "at" : "noon" },
        { "in" : "july", "on" : 20, "at" : "17:00" }
      ]
    }
  }
  ...
}
```

或者：

```
{
  ...
  "trigger" : {
    "schedule" : {
      "yearly" : {
        "in" : [ "jan", "dec" ],
        "on" : [ 10, 20 ],
        "at" : [ "midnight", "noon" ]
      }
    }
  }
  ...
}
```

(6) cron (表达式时刻表)

cron 表达式的元素说明见表 9-2。

表 9-2 cron 表达式元素说明

域名称	是否必须	字段返回	特殊字符
seconds	yes	0-59	, - * /
minutes	yes	0-59	, - * /
hours	yes	0-23	, - * /
day_of_month	yes	1-31	, - * / ? L W
month	yes	1-12 or JAN-DEC	, - * /
day_of_week	yes	1-7 or SUN-SAT	, - * / ? L #
year	no	empty or 1970-2099	, - * /

特殊字符的含义是：

- * 表示匹配该域的任意值, 假如在 minutes 域使用 *, 即表示每分钟都会触发事件。
- ? 只能用在 day_of_month 和 day_of_week 两个域。它也匹配域的任意值, 但实际不会。因为 day_of_month 和 day_of_week 会相互影响。例如想在每月的 20 日触发调度, 不管 20 日到底是星期几, 则只能使用如下写法: 13 13 15 20 * ?, 其中最后一位只能用 ?, 而不能使用 *, 如果使用 * 表示不管星期几都会触发, 实际上并不是这样。
- 表示范围, 例如在 minutes 域使用 5-20, 表示从 5 分到 20 分钟每分钟触发一次。
- / 表示起始时间开始触发, 然后每隔固定时间触发一次, 例如在 minutes 域使用 5/20, 则意味着 5 分钟触发一次, 而 25, 45 等分别触发一次。
- , 表示列出枚举值。例如: 在 minutes 域使用 5,20, 则意味着在 5 分钟和 20 分钟触发一次。
- L 表示最后, 只能出现在 day_of_month 和 day_of_week 域, 如果在 day_of_week 域使用 5L, 意味着在最后的一个星期四触发。
- W 表示有效工作日 (周一到周五), 只能出现在 day_of_month 域, 系统将在离指定日期的最近的有效工作日触发事件。例如: 在 day_of_month 使用 5W, 如果 5 日是星期六, 则将在最近的工作日: 星期五, 即 4 日触发。如果 5 日是星期天, 则在 6 日 (周一) 触发; 如果 5 日在星期一到星期五中的一天, 则就在 5 日触发。另外一点, W 的最近寻找不会跨过月份。
- LW 这两个字符可以连用, 表示在某个月最后一个工作日, 即最后一个星期五。
- # 用于确定每个月第几个星期几, 只能出现在 day_of_month 域。例如在 4#2, 表示某月的第二个星期三。星期是从周日开始作为第一天。

例如下面的表达式表示每天中午 12 点执行:

```
{
  ...
  "trigger" : {
    "schedule" : {
      "cron" : "0 0 12 * * ?"
    }
  }
  ...
}
```

(7) interval 系统内置表达式:

系统内置表达式可以设置 seconds、minutes、hours、days、weeks:

"Xs" 表示每多少秒执行一次。例如: "30s" 表示每 30 秒执行一次。

"Xm" 表示每多少分钟执行一次。例如: "5m" 表示每 5 分钟执行一次。

"Xh" 表示每多少小时执行一次。例如: "12h" 表示每 12 小时执行一次。

"Xd" 表示每多少天执行一次。例如: "3d" 表示每 3 天执行一次。

"Xw" 表示每多少周执行一次。例如: "2w" 表示每 2 周执行一次。

例如下面的例子表示每 5 分钟执行一次：

```
{
  ...
  "trigger" : {
    "schedule" : { "interval" : "5m" }
  }
  ...
}
```

2. Inputs

输入将静态数据加载到告警执行上下文中作为判断的内容，系统支持四种类型输入源：

simple、search、http、chain。

(1) simple Input (简单输入)

简单输入是指将静态的内容加载到告警执行的上下文中，可以定义一个字符串 (STR)、数值 (NUM)，或一个对象 (目标) 等静态数据作为输入源。例如，下面的告警使用了简单的输入来设置一个提醒电子邮件，在中午的每一天执行：

```
{
  "trigger" : {
    "schedule" : {
      "daily" : { "at" : "noon" }
    }
  },
  "input" : {
    "simple" : { "name" : "John" } // 告警中的内容是个简单的单词 John
  },
  "actions" : {
    "reminder_email" : {
      "email" : {
        "to" : "to@host.domain",
        "subject" : "Reminder",
        "body" : "Dear {{ctx.payload.name}}, by the time you read these lines, I'll be gone"
      }
    }
  }
}
```

(2) search Input (搜索输入)

搜索输入指通过 Elasticsearch 的搜索结果作为输入加载到告警执行上下文中作为判断的内容。Conditions、Transforms、Actions 都可以通过结果属性访问搜索中的内容。

例如：

ctx.payload.hits 可以表示搜索的所有内容。

ctx.payload.hits.total 可以表示搜索的结果条数。

ctx.payload.hits.hits.2 可以表示搜索结果的第三条数据。

下面表示搜索所有的日志作为输入的示例：

```
"input" : {
  "search" : {
    "request" : {
      "indices" : [ "logs" ],
      "types" : [ "event" ],
      "body" : { "query" : { "match_all" : {} } }
    }
  }
}
```

下面表示通过模板搜索的日志作为输入的示例：

```
{
  "input" : {
    "search" : { // 通过搜索的结果作为告警的内容
      "request" : {
        "indices" : [ "logs" ],
        "template" : {
          "id" : "my_template",
          "params" : { "value" : 23 }
        }
      }
    }
  }
}
```

(3) HTTP Input (HTTP 协议输入)

HTTP 协议输入指通过 HTTP 请求 Elasticsearch 得到的返回结果作为输入加载到告警执行上下文中作为判断的内容。例如：

```
"input" : {
  "http" : {
    "request" : {
      "host" : "example.com",
      "port" : 9200,
      "path" : "/idx/_search"
    }
  }
}
```

或者用过 DSL 语言进行查询：

```
"input" : {
  "http" : {
    "request" : {
      "host" : "host.domain",
      "port" : 9200,
      "path" : "/idx/_search",
      "body" : "{ \"query\" : { \"match\" : { \"category\" : \"event\" } } }"
```

```

    }
  }
}

```

或者通过模板来进行查询:

```

 : {
  "http" : {
    "request" : {
      "host" : "host.domain",
      "port" : 9200,
      "path" : "/{{ctx.watch_id}}/_search",
      "body" : "{\"query\" : {\"range\" : {\"@timestamp\" : {\"from\" : \"{{ctx.trigger.triggered_time}}\" -5m, \"to\" : \"{{ctx.trigger.triggered_time}}\"}}}}\"
    }
  }
}

```

也可以通过调用 Elasticsearch API 来进行查询:

```

 : {
  "http" : {
    "request" : {
      "host" : "host.domain",
      "port" : "9200",
      "path" : "/_cluster/stats",
      "params" : {"human" : "true"}
    }
  }
}

```

(4) chain Input (链式输入)

链式输入是指可以一次把多个输入加载到告警执行上下文中作为判断的内容。当基于来自多个源的数据执行操作时,链输入是有用的。还可以使一个输入收集的数据是从另一个源加载数据。例如,以下链输入数据请求的路径来自于 HTTP 服务器的一个简单请求:

```

{
  "input" : {
    "chain" : {
      "inputs" : [ // 将一个链中的输入指定为一个数组,以保证输入的顺序处理
        { "first" : { "simple" : { "path" : "/_search" } } },
        {
          "second" : { "http" : { "request" : { "host" : "localhost", "port" : 9200,
            "path" : "{{ctx.payload.first.path}}" } } } // 加载由第一个输入设置的路径
        }
      ]
    }
  }
  ...
}

```

3. Condition 所有的日志作为输入的示例:

当一个告警被触发时, Condition 配置的条件是否满足决定了是否执行下一步的动作。系统支持四种类型的条件类型: always、never、script、compare。

always 表示触发时间到后始终要执行。例如:

```
"condition" : {"always" : {}}
```

never 表示永远不会被执行, 这种情况通常只用于测试。例如:

```
PUT _watcher/watch/my-watch
```

```
{
  ...
  "condition" : {"never" : {}}
  ...
}
```

script 表示用一个脚本来表示告警的条件。默认脚本语言 Groovy。可以使用任何 Elasticsearch 支持的脚本语言。以下代码片段将使用内联脚本的情况总是返回 true:

```
"condition" : {
  "script" : "return true"
}
```

可以使用文件作为脚本数据源, 例如:

```
"condition" : {
  "script" : {
    "file" : "my_script",
    "lang" : "javascript",
    "params" : {"result" : true}
  }
}
```

compare 表示一种简单的将执行上下文模型中的值与给定值进行比较的告警条件。例如下面的例子表示查询的结果条数大于等于 5 的时候满足条件:

```
{
  ...
  "condition" : {
    "compare" : {
      "ctx.payload.hits.total" : { "gte" : 5 }
    }
  }
  ...
}
```

条件有以下几种: eq 表示等于, not_eq 表示不等于, gt 表示大于, gte 表示大于等于, lt 表示小于, lte 表示小于等于。

例如:

```
{
```



```

...
"condition" : {
  "compare" : {
    "ctx.payload.aggregations.status.buckets.error.doc_count" : {
      "not_eq" : "{ctx.payload.aggregations.handled.buckets.true.doc_count}"
    }
  }
}
...
}

```

4. Transform

Transform 这项不是必须的，表示在输入到执行动作的过程中进行的数据处理。可以定义在两个地方的变换：

- ❑ 在定义告警的顶级结构中，在这种情况下，在执行任何告警操作之前被转换。
- ❑ 作为一个特定行动定义的一部分。在这种情况下，执行该操作之前被转换。转换仅适用于特定的动作转换。

例如下面的示例定义了两个转换，一个是全局转换和一个在 my_webhook 行为中进行的转换。

```

{
  "trigger" : { ... }
  "input" : { ... },
  "condition" : { ... },
  "transform" : {
    "search" : { "body" : { "query" : { "match_all" : {} } } }
  }
  "actions" : {
    "my_webhook" : {
      "transform" : { "script" : "return ctx.payload.hits" }
      "webhook" : {
        "host" : "host.domain",
        "port" : 8089,
        "path" : "/notify/{ctx.watch_id}"
      }
    }
  }
}
...

```

转换支持搜索转换、脚本转换和链转换，分别用 search、script、chain 表示。

search 转换是在群集上执行搜索的转换，并用返回的搜索结果替换告警执行上下文中的内容，例如：

```

{
  "transform" : {
    "search" : { // 表示一次搜索转换

```

```

3. Cor
    "request" : {
      "indices" : [ "logstash-*" ],
      "body" : {
        "size" : 0,
        "query" : { "match" : { "priority" : "error" } }
      }
    }
  }
}

```

script 转换是在群集上执行脚本的转换，并用返回的脚本执行结果替换告警执行上下文中的内容，例如：

```

{
  ...
  "transform" : { "script" : "return [ time : ctx.trigger.scheduled_time ]" }
  ...
}

```

链式 (chain) 转换是指可以一次把多个转换加载到转换中，利用它可以建立更复杂的转换。例如，可以将搜索转换和脚本转换同时使用，见下面的示例：

```

"transform" : {
  "chain" : [
    {
      "search" : { // 条件一：搜索转换
        "indices" : [ "logstash-*" ],
        "body" : {
          "size" : 0,
          "query" : { "match" : { "priority" : "error" } }
        }
      },
      { // 条件二：脚本转换
        "script" : "return [ error_count : ctx.payload.hits.total ]"
      }
    ]
  }
}

```

5. Action

Action 表示当满足告警条件时会发生的动作。告警的动作可以是邮件、HipChat、Slack 等，例如：

请求：PUT http://127.0.0.1:9200/_watcher/watch/log_event_watch

```

{
  "metadata" : { "color" : "red" },
  "trigger" : {
    "schedule" : { "interval" : "5m" }
  },

```

```

 : {
  "search" : {
    "request" : {
      "indices" : "log-events",
      "body" : { "size" : 0, "query" : { "match" : { "status" : "error" } } }
    }
  },
  "condition" : {
    "script" : "return ctx.payload.hits.total > 5"
  },
  "actions" : {
    "email_administrator" : {
      "throttle_period": "15m", // 表示触发动作的周期在 15 分钟，以减少告警的数量。
      "email" : {
        "to" : "sys.admin@host.domain",
        "subject" : "Encountered {{ctx.payload.hits.total}} errors",
        "body" : "Too many error in the system, see attached data",
        "attachments" : {
          "attached_data" : {
            "data" : {
              "format" : "json"
            }
          }
        }
      },
      "priority" : "high"
    }
  }
}

```

图 9-1 说明了在执行过程中的每一个告警的每一个动作所做的细节。

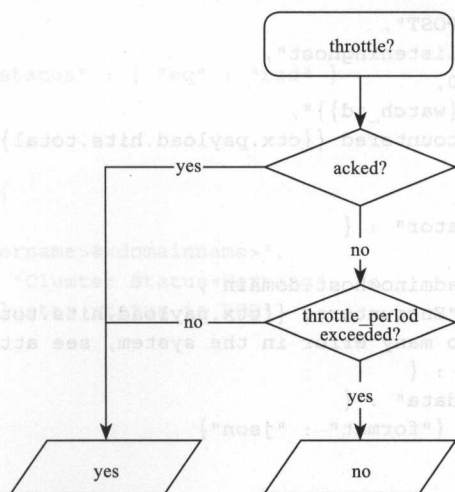


图 9-1 Action 告警流程图

9.1.3 示例

下面的代码片段显示了用来寻找记录错误事件的一个告警的定义：

请求：PUT http://127.0.0.1:9200/_watcher/watch/log_event_watch

```
{
  "metadata" : { ① "color" : "red"},
  "trigger" : { ②
    "schedule" : {"interval" : "5m"}
  },
  "input" : { ③
    "search" : {
      "request" : {
        "indices" : "log-events",
        "body" : {
          "size" : 0, "query" : { "match" : { "status" : "error" } } }
        }
      }
    },
    "condition" : { ④
      "script" : "return ctx.payload.hits.total > 5"
    },
    "transform" : { ⑤
      "search" : {
        "request" : {
          "indices" : "log-events",
          "body" : { "query" : { "match" : { "status" : "error" } } }
        }
      }
    }
  },
  "actions" : { ⑥
    "my_webhook" : {
      "webhook" : {
        "method" : "POST",
        "host" : "mylisteninghost",
        "port" : 9200,
        "path" : "/" + {{watch_id}} + "/",
        "body" : "Encountered {{ctx.payload.hits.total}} errors"
      }
    },
    "email_administrator" : {
      "email" : {
        "to" : "sys.admin@host.domain",
        "subject" : "Encountered {{ctx.payload.hits.total}} errors",
        "body" : "Too many error in the system, see attached data",
        "attachment" : {
          "attached_data" : {
            "data" : { "format" : "json" }
          }
        }
      }
    },
    "priority" : "high"
  }
}
```

```

    }
  }
}

```

说明如下：

- ①元数据——可以将可选的静态元数据附加到一个告警上。
- ②触发——这个触发计划表示每 5 分钟执行一次。
- ③输入——搜索日志事件索引中的错误事件，并将响应加载到告警的内容中。
- ④条件——检查是否有超过 5 个错误事件。如果有，继续执行下面内容。
- ⑤转换——如果满足告警条件，所有的动作都访问这个转换。
- ⑥动作——这个告警有两个动作。my_webhook 将这个问题通知到第三方系统。email_administrator 操作发送邮件到系统管理员，在电子邮件中包含错误内容。

下面的代码片段显示了用来检查集群健康状态的一个告警的定义。

请求：PUT http://127.0.0.1:9200/_watcher/watch/cluster_health_watch

```

{
  "trigger" : { ①
    "schedule" : { "interval" : "10s" }
  },
  "input" : { ②
    "http" : {
      "request" : {
        "host" : "127.0.0.1",
        "port" : 9200,
        "path" : "/_cluster/health"
      }
    }
  },
  "condition" : { ③
    "compare" : {
      "ctx.payload.status" : { "eq" : "red" }
    }
  },
  "actions" : { ④
    "send_email" : {
      "email" : {
        "to" : "<username>@<domainname>",
        "subject" : "Cluster Status Warning",
        "body" : "Cluster status is RED"
      }
    }
  }
}

```

说明如下：

- ①设置监控告警的频率每 10 秒监控一次。

- ② 设置输入获取需要分析的数据。
- ③ 增加一个条件, 评估健康状态来决定是否需要采取行动。
- ④ 如果集群健康状态符合条件, 执行发送邮件的操作。

9.1.4 告警输出配置

告警输出可以为邮件、Webhook、Logging、HipChat、Slack、PagerDuty。

1. 邮件配置

必须在 `elasticsearch.yml` 中配置邮件账户并且重启 Elasticsearch 进程:

```
watcher.actions.email.service.account:
work:
```

```
  profile: gmail
  email_defaults:
    from: <email>
  smtp:
    auth: true
    starttls.enable: true
    host: smtp.gmail.com
    port: 587
    user: <username>
    password: <password>
```

在 Actions 中使用如下:

```
"actions" : {
  "email_admin" : {
    "email": {
      "to": "'John Doe <john.doe@example.com>'",
      "attachments" : {
        "my_report.pdf" : { // 附件的类型
          "http" : { // 附件的来源
            "content_type" : "application/pdf",
            "request" : {"url": "http://example.org/foo/my-report" }
          }
        },
        "data.yml" : { // 数据的格式, 默认是 JSON 格式
          "data" : {"format" : "yaml" }
        }
      }
    }
  }
}
```

2. webhook

webhook 主要是提交 Web 数据, 这个不需要单独配置, 只需要在 Actions 中配置, 例如:

```
"actions" : {
  "my_webhook" : {
```

```

"transform" : { ... },
"throttle_period" : "5m",
"webhook" : {
  "method" : "POST", // 方法
  "host" : "mylisteningserver", // 主机
  "port" : 9200, // 端口
  "path" : ":{ctx.watch_id}", // 路径
  "body" : "{ctx.watch_id}:{ctx.payload.hits.total}" // 内容
}
}
}

```

3. Logging

Logging 主要是记录日志到 Elasticsearch 中，这个不需要单独配置，只需要在 Actions 中配置，例如。

```

"actions" : {
  "log" : { // 关键字
    "transform" : { ... },
    "logging" : {
      "text" : "executed at {{ctx.execution_time}}" // 日志内容
    }
  }
}

```

4. HipChat、Slack、PagerDuty

这三个在国内用的很少，就不介绍了，需要用时请查阅相关资料。

9.1.5 告警管理

前面介绍了告警的新增操作，告警的管理中还有列出目前现有的告警，和删除已经存在的告警。

1. 列出告警

可以通过搜索索引 .watches 列出所有告警的配置，例如：

请求：GET http://127.0.0.1:9200/.watches/_search

```

{
  "fields" : [],
  "query" : { "match_all" : { } }
}

```

2. 删除告警

请求：DELETE http://127.0.0.1:9200/_watcher/watch/cluster_health_watch

返回值：

```

{"_id": "my-watch", "_version": 8, "found": true}

```

9.2 监控

Marvel 是商业监控方案，用来监控 Elasticsearch 集群历史状态的有力工具，便于性能优化以及故障诊断。监控主要分为六个层面，分别是集群层、节点层、索引层、分片层、事件层、Sense。

9.2.1 安装

Marvel 是以 Elasticsearch 插件的形式存在的。在最新版本中 Marvel 有两个组件，一个是代理，需要安装在 Elasticsearch 集群中，一个是 Kibana 的插件，需要安装在 Kibana 中。安装步骤如下：

□ 在 Elasticsearch 中安装 Marvel 插件：

```
bin/plugin install license
bin/plugin install marvel-agent
```

□ 在 Kibana 中安装 Marvel 插件：

```
bin/kibana plugin --install elasticsearch/marvel/latest
```

□ 启动 Elasticsearch 和 Kibana：

```
bin/elasticsearch
bin/kibana
```

安装好后在浏览器中输入：<http://127.0.0.1:5601/> 即可看到界面如图 9-2 所示。

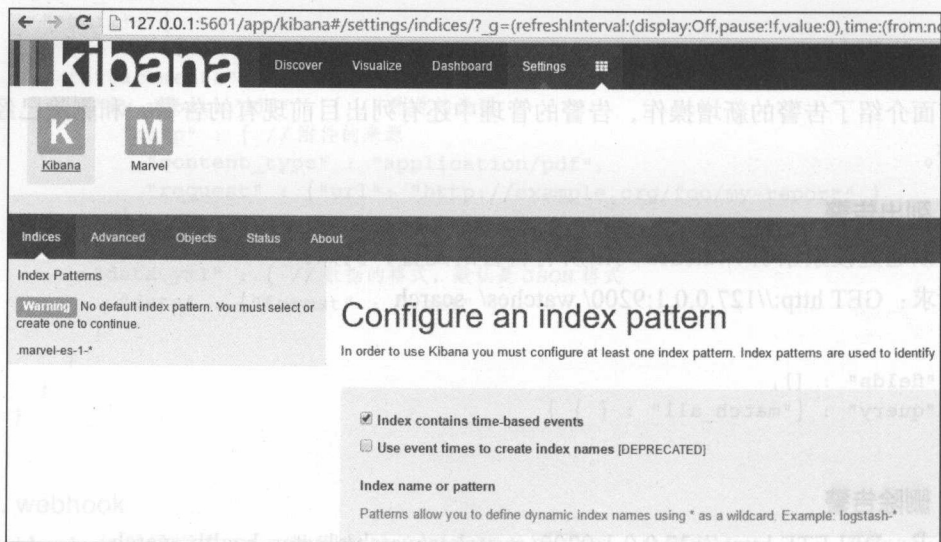


图 9-2 Kibana 首页

然后在界面的菜单中选择 Marvel，会看到如图 9-3 所示的界面。

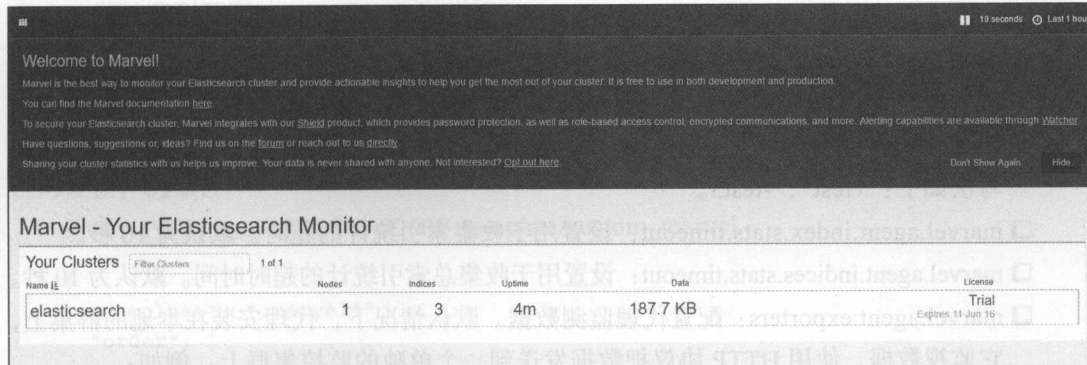


图 9-3 Marvel 首页

然后点击 elasticsearch，看到如图 9-4 所示的监控界面。



图 9-4 Marvel 监控页

9.2.2 配置

1. 监控参数配置

可以通过在每个节点的 `elasticsearch.yml` 中配置 Marvel 参数来控制从 Elasticsearch 集群中采集数据，可以添加一个自定义索引模板来更改索引监控的设置，从一个集群收集的数据中创建存储。

监控设置的参数在 `elasticsearch.yml` 文件中，从 `marvel.agent` 开始：

- ❑ `marvel.agent.cluster.state.timeout`：设置用于收集集群状态的超时时间。默认为 10 秒。

- ❑ `marvel.agent.cluster.stats.timeout`: 设置用于收集集群统计的超时时间。默认为 10 秒。
- ❑ `marvel.agent.indices`: 控制哪些索引数据被收集, 默认所有索引。指定索引的名称可以用逗号分隔, 例如 `test1, test2, test3`。名称中可以包含通配符, 例如 `test*`。可以明确地包括或排除相关索引。例如, 包括所有以 `test` 开头, 但要排除 `test3` 的索引, 写法如下: `+test*, -test3`。

- ❑ `marvel.agent.index.stats.timeout`: 设置用于收集索引统计的超时。默认为 10 秒。
- ❑ `marvel.agent.indices.stats.timeout`: 设置用于收集总索引统计的超时时间。默认为 10 秒。
- ❑ `marvel.agent.exporters`: 配置代理监测数据。默认情况下, 代理安装在本地的群集上, 它监视数据, 使用 HTTP 协议把数据发送到一个单独的监控集群上, 例如:

```
marvel.agent.exporters:
  id1:
    type: local
    // 默认为监控本地节点

  id2:
    type: http
    host: [ "http://domain:port", ... ]
    // 通过 HTTP 协议监控
    // 类型有 local 或者 HTTP
    // 请求的路径, 可以是 HTTPS 协议

    auth:
      username: <string>
      password: <string>
      // 认证的用户名
      // 认证的密码

    connection:
      timeout: <time_value>
      read_timeout: <time_value>
      keep_alive: true | false
      // 超时时间 (默认为 6 秒)
      // 响应时间 (默认为超时时间的 10 倍)
      // 是否长连接 (默认为 true)

    ssl:
      hostname_verification: true | false // 是否检查证书 (默认为 true)
      protocol: <string>
      // 证书协议 (默认为 TLSv1.2)
      truststore.path: /path/to/file
      // 信任库的路径
      truststore.password: <string>
      // 信任库的密码
      truststore.algorithm: <string>
      // 信任库的格式 (默认为 SunX509)
```

```
index:
  name:
    time_format: <string>
    // 后缀时间格式 (默认为 "YYYY.MM.dd")
```

- ❑ `marvel.agent.index.recovery.active_only`: 控制是否所有的恢复数据被收集。设置为 `true` 收集有效的恢复数据。默认为 `false`。
- ❑ `marvel.agent.index.recovery.timeout`: 设置用于收集恢复数据的超时时间。默认为 10 秒。
- ❑ `marvel.agent.interval`: 控制收集数据样本的频率。默认设置为 10 秒, 设置为 -1 表示禁用数据收集。
- ❑ `marvel.history.duration`: 设置监控创建的索引将被自动删除的保留时间。默认为 7 天。设置为 -1 表示禁用自动删除监控索引。

2. 监控索引配置

监控使用一个索引模板来配置用于存储从集群中收集的数据。

可以通过如下命令搜索默认模板: `GET /_template/.marvel-es`

默认情况下, 模板配置一个分片和一个复制的监控索引。要覆盖默认设置, 可以添加自己的模板, 例如:

请求: `PUT http://127.0.0.1:9200/_template/custom_marvel`

```
{
  "template": ".marvel*",
  "order": 1,
  "settings": {
    "number_of_shards": 5,
    "number_of_replicas": 2
  }
}
```

3. Kibana 配置相关

在 Kibana 的配置文件 (`kibana.yml`) 中可以设置 Marvel 监控参数, 在大多数情况下, 默认值已经可以很好地工作。

- ❑ `marvel.max_bucket_size`: 从搜索索引节点指标聚合中返回执行聚合搜索索引和节点的桶的数量。默认值为 10000。
- ❑ `marvel.min_interval_seconds`: 时间桶所能表示的最小秒数。默认值为 10 秒。
- ❑ `marvel.node_resolver`: 被认为是唯一一节点的标志。
- ❑ `marvel.report_stats`: 是否将集群统计数据发送到 Elastic 公司。默认值为 `true`。

4. Tribe 部落节点监控配置

如果你通过一个部落节点连接到一个集群, 需要在部落节点以及集群中的节点上安装监控代理来监控集群。如果群集上设置了权限, 则还需要在部落节点上安装和配置权限。

排除部落节点时需要在 `elasticsearch.yml` 配置文件中设置 `marvel.enabled` 为 `false`, 例如:

```
node.name: tribe
marvel.enabled: false
tribe:
  tl:
    cluster.name: cluster1
    discovery.zen.ping.unicast.hosts: ["cluster1-node1:9300", "cluster1-node2:9300"]
```

有了这个配置后, 在监控用户界面中显示的节点数将包含部落节点, 但节点列表并不包含部落节点, 因为没有将任何数据导出到监视群集中。

下面的示例是在监控所有部落级的数据:

```
node.name: tribe
marvel.enabled: false
```

```

tribe:
  t1:
    cluster.name: cluster1
    discovery.zen.ping.unicast.hosts: ["cluster1-node1:9300", "cluster1-node2:9300"]
    marvel:
      enabled: true
      agent.exporters:
        id1:
          type: http
          host: ["monitoringhost:9200"]

```

当启用部落节点的数据收集时，它包含在节点计数和节点列表中。值得注意的是，部落节点只支持 HTTP 的输出，部落节点的数据必须发送到外部监控群集上。

9.3 权限管理

Shield 是商业权限管理插件，它可以保护 Elasticsearch 中的数据，采用加密的通信密码，基于角色的访问控制，IP 过滤和审计等。

Shield 是以 Elasticsearch 插件的形式存在的。安装过程如下：

1) 在 Elasticsearch 中安装 Shield 插件：

```
bin/plugin install license
```

```
bin/plugin install shield
```

2) 启动 Elasticsearch：

```
bin/elasticsearch
```

3) 添加管理员账号：

```
bin/shield/esusers useradd es_admin -r admin
```

4) 测试是否生效。在页面访问数据的时候，会显示如图 9-5 所示的界面。

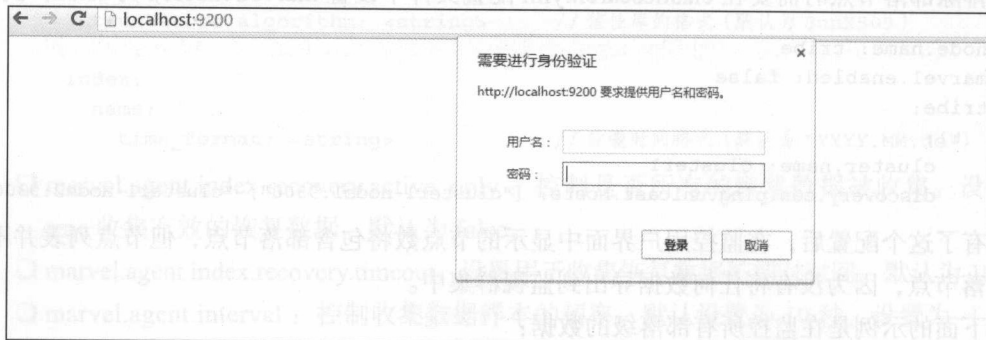


图 9-5 Shield 登录页面

当输入正确的账户和密码的时候，才可以访问数据。

9.3.1 工作原理

Shield 是 Elasticsearch 的一个插件，一旦安装完成，插件将会拦截所有 API 请求，然后对请求进行认证和授权的校验。该插件同时提供 SSL 安全协议来传输网络数据，该插件提供了审计日志记录的能力，用来进行验证和审计。

1. 用户认证

Shield 定义了一组已知的用户，以便对请求的用户进行身份验证。这些用户的集合被定义为一个抽象的领域。这个抽象的领域可以是数据库、本地文件、LDAP、活动目录或者 KPI。

2. 授权

Shield 的访问授权的数据模型由以下几个因素构成：

- 资源，包括集群、索引、别名等。
- 权限，对资源的一个或者多个操作。
- 许可，对应安全资源的一个或者多个权限。有 2 种类型的许可：集群和索引。
- 角色，权限的集合。
- 用户，准许访问资源的对象。

3. 节点认证和信道加密

Shield 可以使用 SSL / TLS 包进行内部通信。当 SSL / TLS 启用，节点相互交换验证的证书，建立节点之间的信任，验证防止未经身份验证的节点加入集群，通过验证后，内部通信是经过加密的。

4. IP 过滤

Shield 提供基于 IP 的针对节点的访问控制。通过 IP 控制可以限制其他机器访问 Elasticsearch 服务器，可以设置白名单或者黑名单进行过滤，可以设置 IP 或者网段。

5. 审计

安全审计功能提供了访问 Elasticsearch 特定事件或者活动的记录，包括登录，授权，拒绝访问等。

9.3.2 用户认证

1. 用户认证方式

系统支持以下几种方式的用户认证：

- Native：一个内置的本地认证系统，默认可用。
- File：一种内置的基于文件的认证系统，默认可用。
- LDAP：通过外部轻量级目录协议进行身份验证。

□ AD: 通过外部活动目录服务的身份验证。

□ PKI: 通过使用可信的 X.509 证书的认证。

Native、File、LDAP、AD 这四种方式是用用户密码进行认证的。系统也支持自定义用户认证方式,可以同时存在多种认证方式,它本质上是一个配置的认证方式的优先列表。列表的顺序决定了将要认证的顺序。在认证过程中,系统将尝试一次验证一个认证方式的请求。一旦一种认证方式认证成功,通过身份认证的用户将与请求进行相关联(下一步是授权阶段)。如果一个认证方式无法对请求进行身份验证,则用认证链中的下一个认证方式进行验证。如果链中所有的认证都没有通过,认证就失败了,将返回错误(HTTP 状态代码 401)。

例如下面的代码片段配置认证链包含文件认证和本地服务器认证,以及两个 LDAP 和 Active Directory 域服务器认证:

```
shield.authc:
  realms:
    native:
      type: native
      order: 0
    file:
      type: file
      order: 1
    ldap1:
      type: ldap
      order: 2
      enabled: false
      url: 'url_to_ldap1'
      ...
    ldap2:
      type: ldap
      order: 3
      url: 'url_to_ldap2'
      ...
    ad1:
      type: active_directory
      order: 4
      url: 'url_to_ad'
```

2. 匿名用户访问

认证过程可以分为两个阶段:令牌提取和用户认证。在第一阶段(令牌提取阶段),配置的服务器将请求尝试从传入的请求中提取/解析身份验证令牌。找到身份验证的令牌,然后将令牌用于认证。在没有任何身份验证令牌的情况下,传入的请求被认为是匿名的。

默认情况下,匿名请求被拒绝,并返回一个身份验证错误码(状态代码 401)。可以通过配置改变这种行为,在 `elasticsearch.yml` 文件中配置下面的内容:

```
shield.authc:
  anonymous:
```

```

username: anonymous_user
roles: role1, role2
authz_exception: true

```

3. 本地认证

本地认证配置如下:

```

shield:
  authc:
    realms:
      native:
        type: native
        order: 0

```

可以通过 Users API 来完成本地认证用户的管理。

增加本地用户如下所示:

请求: POST http://127.0.0.1:9200/_shield/user/ironman

```

{
  "password" : "j@rV1s", // 密码, 必选
  "roles" : [ "admin", "other_role1" ], // 用户角色, 必选
  "full_name" : "Tony Stark", // 用户全名, 可选
  "email" : "tony@starkcorp.co", // 用户邮箱, 可选
  "metadata" : { // 与用户关联的元数据, 可选
    "intelligence" : 7
  }
}

```

查询本地用户列表如下所示:

请求: GET http://127.0.0.1:9200/_shield/user

返回值:

```

{
  "found" : true,
  "users" : [
    {
      "username" : "ironman",
      "roles" : [ "admin", "other_role1" ],
      "full_name" : "Tony Stark",
      "email" : "tony@starkcorp.co",
      "metadata" : {
        "intelligence" : 7
      }
    }
  ]
}

```

删除本地用户如下:


```
DELETE /_shield/user/ironman
```

4. LDAP 认证

可以通过配置使用目录访问协议 (LDAP) 进行用户认证。需要配置 LDAP 域和指定 LDAP 的角色在角色映射文件中。为了保护密码, 与 LDAP 服务器之间的通信必须使用 SSL / TLS 加密。客户节点通过 SSL / TLS 连接到 LDAP 服务器需要 LDAP 服务器的证书或服务器的根 CA 证书安装在自己的密钥库和信任存储区中。在 `elasticsearch.yml` 中添加 `shield.authc.realms` 命名空间来进行配置, 例如, 下面的代码片段显示了在 LDAP 中配置用户认证:

```
shield:
  authc:
    realms:
      ldap1:
        type: ldap
        order: 0
        url: "ldaps://ldap.example.com:636"
        bind_dn: "cn=ldapuser, ou=users, o=services, dc=example, dc=com"
        bind_password: changeme
        user_search:
          base_dn: "dc=example,dc=com"
          attribute: cn
        group_search:
          base_dn: "dc=example,dc=com"
        files:
          role_mapping: "CONFIG_DIR/shield/role_mapping.yml"
          unmapped_groups_as_roles: false
```

如果你的 LDAP 环境使用一些特定的标准命名条件的用户, 可以使用用户 DN 模板来配置。这种方法的优点是搜索不一定要找出用户的 DN。但多个绑定操作可能需要找到正确的用户 DN。例如:

```
shield:
  authc:
    realms:
      ldap1:
        type: ldap
        order: 0
        url: "ldaps://ldap.example.com:636"
        user_dn_templates:
          - "cn={0}, ou=users, o=marketing, dc=example, dc=com"
          - "cn={0}, ou=users, o=engineering, dc=example, dc=com"
        group_search:
          base_dn: "dc=example,dc=com"
        files:
          role_mapping: "/mnt/elasticsearch/group_to_role_mapping.yml"
          unmapped_groups_as_roles: false
```

存储在每个节点的角色映射文件指定了 LDAP 组。当一个用户使用 LDAP 认证时, 用

户所属的组的角色定义了用户的权限。例如，下面的映射配置指定了 LDAP 管理员组和普通用户组：

```
monitoring:
- "cn=admins,dc=example,dc=com"
user:
- "cn=users,dc=example,dc=com"
- "cn=admins,dc=example,dc=com"
```

SSL / TLS 加密配置方法如下：

1) 首先要生成证书：

```
cd CONFIG_DIR/shield
keytool -importcert -keystore node01.jks -file cacert.pem -alias ldap_ca
```

2) 在 `elasticsearch.yml` 配置文件中配置 SSL / TLS 加密：

```
shield.ssl.keystore.path: /home/es/config/shield/node01.jks
shield.ssl.keystore.password: myPass
shield.ssl.keystore.key_password: myKeyPass
```

3) 配置 LDAPS 协议指定的 URL 属性和端口号。例如，url: ldaps://ldap.example.com:636。

4) 重启 Elasticsearch 使 `elasticsearch.yml` 生效。

5. AD 认证

可以通过配置使用活动目录 (AP) 进行用户认证。需要配置 AP 域和指定 AP 的角色在角色映射文件中。为了保护密码，和 AP 服务器之间的通信必须使用 SSL / TLS 加密。客户节点通过 SSL / TLS 连接到 AP 服务器需要 AP 服务器的证书或服务器的根 CA 证书安装在自己的密钥库和信任存储区中。在 `elasticsearch.yml` 中添加 `shield.authc.realms` 命名空间来进行配置，例如，下面的代码片段显示了在 AD 中配置用户认证：

```
shield:
  authc:
    realms:
      active_directory:
        type: active_directory
        order: 0
        domain_name: ad.example.com
        url: ldaps://ad.example.com:636
        unmapped_groups_as_roles: true
```

AD 认证的很多内容和 LDAP 基本一样，其他内容请参考上一节。

6. PKI 身份认证

使用公共密钥基础设施 (PKI) 证书对用户进行身份验证要求客户提供 X.509 证书。可以结合使用 PKI 认证和用户名 / 密码认证。例如，可以启用 SSL / TLS 在传输层上使用一个 PKI 验证的 X.509 证书，同时还验证 HTTP 协议中使用的用户名和密码。还可以设置 `shield.transport.ssl.client.auth` 允许客户无证书认证等。在 `elasticsearch.yml` 中添加 `shield.authc`。

realms 命名空间来进行配置，例如：

```
shield:
  authc:
    realms:
      pki1:
        type: pki
```

分配给 PKI 用户的角色映射文件存储在每个节点，可以为 PKI 用户分配角色，在证书中区分不同的用户。例如，下面的映射配置指定 John Doe 的用户角色：

```
user:
  - "cn=John Doe,ou=example,o=com"
```

7. 基于文件的授权

管理和认证基于文件的认证是系统内置的认证方式。可以通过 esusers 控制台命令来增加和删除用户，分配用户角色，管理用户密码。在 elasticsearch.yml 中添加 shield.authc.realms 命名空间来进行配置，例如：

```
shield:
  authc:
    realms:
      file1:
        type: file
        order: 0
```

用户管理命令在 ES_HOME/bin/shield 目录下。

增加用户：esusers useradd <username>

增加用户同时设置密码：esusers useradd <username> -p <secret>

增加用户同时设置角色：esusers useradd <username> -r <comma-separated list of role names>

查询用户：esusers list

管理密码：esusers passwd <username> -p <password>

用户分配角色：用 -a 增加角色，-r 删除角色。例如：

```
esusers roles <username> -a <comma-separated list of roles> -r <comma-separated list of roles>
```

删除用户：userdel <username>

9.3.3 角色管理

Shield 提供了基于角色的访问控制 (RBAC)，这种方式可以控制用户在一个 Elasticsearch 集群执行哪些行为。默认情况下，所有操作都受到限制。分配给用户角色后，用户才可以执行授权内的操作。

1. 增加角色

请求: POST http://127.0.0.1:9200/_shield/role/my_admin_role

```
{
  "cluster": ["all"], // 可以访问的集群, 可选
  "indices": [ // 可以访问的索引, 可选
    {
      "names": [ "index1", "index2" ], // 可以访问的索引
      "privileges": ["all"], // 可以授权的索引
      "fields": [ "title", "body" ], // 可以访问的索引字段, 默认全部
      "query": "{ \"match\": { \"title\": \"foo\"} }" // 可以查询的文档, 默认全部
    }
  ],
  "run_as": [ "other_user" ] // 允许 other_user 用户用这个角色来执行操作, 可选
}
```

2. 查看角色

请求: GET http://127.0.0.1:9200/_shield/role/my_admin_role,log_admin_role

返回值:

```
{
  "found" : true,
  "roles" : [ {
    "name" : "my_admin_role", // 自定义增加的角色
    "cluster" : [ "all" ],
    "indices" : [ {
      "names" : [ "index1", "index2" ],
      "privileges" : [ "all" ],
      "fields" : [ "title", "body" ],
      "query" : "{ \"match\": { \"title\": \"foo\"} }"
    } ],
    "run_as" : [ "other_user" ]
  },
  {
    "name" : "log_admin_role", // 内置的另一个角色
    "cluster" : [ "all" ],
    "indices" : [ {
      "names" : [ "logs-*" ],
      "privileges" : [ "all" ],
    } ]
  } ]
}
```

3. 删除角色

请求: DELETE http://127.0.0.1:9200/_shield/role/my_admin_role

9.3.4 综合示例

本节中的示例演示如何部署 Shield 来保护 Elasticsearch 集群。我们用电子商务为例子，这个例子商店的电子商务商店由以下几个组成部分：

- 一个网上商店的应用程序，需要执行查询操作；
- 每晚批量导入的任务，它要执行 `reindexes` 操作，确保第二天的有正确定价；
- 一个更新机制，在业务时间内在每个文档的基础上同时写入数据；
- 需要访问销售数据索引的销售代表。

定义角色如下：

bulk:

```
indices:
  - names: 'products_*'
  privileges:
    - write
    - read
    - manage
```

updater:

```
indices:
  - names: 'products'
  privileges:
    - write
```

webshop:

```
indices:
  - names: 'products'
  privileges:
    - read
```

monitoring:

```
cluster:
  - monitor
indices:
  - names: '*'
  privileges:
    - monitor
```

sales_rep :

```
cluster:
  - none
indices:
  - names: 'sales_*'
  privileges:
    - all
  - names: 'social_events'
  privileges:
    - read
```


角色说明:

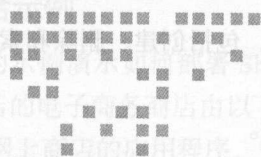
- ❑ bulk 角色可以访问以 products_ 开头的任何索引, 包括创建、删除和索引数据。
 - ❑ updater 角色只能对 products 的索引进行写操作。
 - ❑ webshop 角色只能对 products 的索引进行读操作。
 - ❑ monitoring 角色用于在 Web 应用程序的内部屏幕上显示的监控数据, 可以访问 Monitor 集群中的任何索引。
 - ❑ sales_rep 角色可以访问任何以 sales_ 开头的索引, 并可以读取 social_events 索引。
- 用 esusers 命令来创建用户, 同时对用户进行授权:

```
bin/shield/esusers useradd webshop -r webshop,monitoring
bin/shield/esusers useradd bulk -r bulk
bin/shield/esusers useradd updater -r updater
bin/shield/esusers useradd best_sales_guy_of_the_world -r sales_rep
bin/shield/esusers useradd second_best_sales_guy_of_the_world -r sales_rep
```

随着用户和角色的定义, 现在需要修改应用程序。应用程序的每个部分都必须使用用户名和密码到 Elasticsearch 中进行验证。

9.4 小结

本章介绍了和 Elasticsearch 相关的告警、监控和权限管理, 这几个功能主要解决了默认 Elasticsearch 中没有的或者不方便的地方, 这几个地方都是在管理、运维中非常重要的, 不过遗憾的是这几个插件都是收费的, 在经济允许的情况下可以购买, 也算是对开源软件的一种赞助行为。



通过前面的介绍，知道了 Elasticsearch 有很多强大的功能，但这些功能能用在什么地方呢，本章主要介绍与 Elasticsearch 相关的两个应用。当然市场上已经有越来越多的企业开始使用 Elasticsearch，对 Elasticsearch 的发展非常有帮助。

Logstash：是一个灵活的开放源码的数据收集、处理、传输的工具。Logstash 可以处理日志、事件、非结构化的数据，并把它们输出出来，包括可以输出到 Elasticsearch 中。

Kibana：是一个开源的数据可视化平台，可以把数据以强大的图形化方式展示出来。从柱状图到地图等，它可以通过多个图表的组合来生成更为强大的仪表面板，帮助人们理解、分析和分享数据。

在业界一般把 Elasticsearch+Logstash+Kibana (ELK) 的组合专门用来处理日志，存储日志，展示日志。

10.1 Logstash

注意：关于安装文档，网络上有很多可以参考，而且三件套各自的版本很多，差别也不一样，需要版本匹配上才能使用，推荐直接使用官网的最新版。本示例的版本是 Elasticsearch -2.3.2、Logstash -2.3.2、Kibana -4.5.0 版本。

下面以一个示例介绍 ELK 的使用，这个示例是收集 Nginx 的日志。

Logstash、Elasticsearch、Kibana 三个产品的安装都可以是绿色的，只要安装好 Java 后就可以直接使用。

10.1.1 配置

Logstash 的原理图见图 10-1。

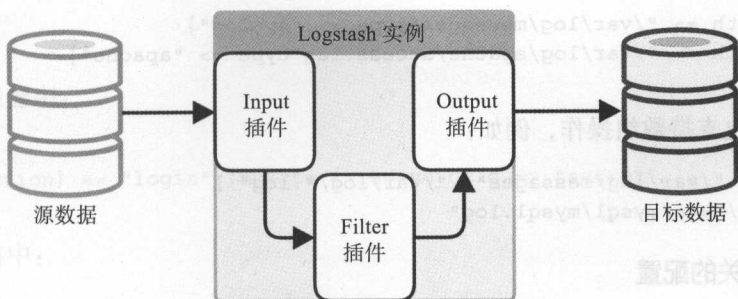


图 10-1 Logstash 的原理图

Input 是收集的数据来源，Output 是数据输出到哪里，Filter 代表过滤规则，表示数据的过滤和转换。这三部分是通过配置文件来完成的。

为了使用 Logstash，首先要创建一个配置文件，配置文件中指定了使用的插件和每个插件的设置。它们的使用要符合一定的标准，当运行 Logstash 时，可以使用 -f 参数指定配置文件启动。

下面我们来创建一个简单的配置文件，命名为 logstash-simple.conf，并保存在 Logstash 的根目录下，内容如下：

```
input { stdin { } }
output {
  elasticsearch { hosts => ["localhost:9200"] }
  stdout { codec => rubydebug }
}
```

启动的时候加上 -f 参数启动：

```
bin/logstash -f logstash-simple.conf
```

1. 配置文件的结构

配置文件由输入 -> 过滤 -> 输出三部分组成，每部分都是由插件构成的，这些插件负责处理日志的不同过程，例如，一个典型的配置如下：

```
input {
  ...
}
filter {
  ...
}
output {
  ...
}
```

一个插件的配置由插件的名称和插件的设置组成。例如，输入部分可以配置两个文件输入：

```
input {
  file {path => "/var/log/messages" type => "syslog"}
  file {path => "/var/log/apache/access.log" type => "apache"}
}
```

同时参数也支持数组操作，例如：

```
path => [ "/var/log/messages", "/var/log/*.log" ]
path => "/data/mysql/mysql.log"
```

2. 事件相关的配置

每个事件都有不同的属性，比如 apache 的访问日志，可以包括状态码、协议、路径、客户端 ip 等，在 Logstash 中这些属性叫作 fields。由于它们是事件属性，所以这些配置选项只会在过滤器和输出块中工作。

字段的属性默认是 JSON 格式的，例如下面有五个字段组成事件：

```
{
  "agent": "Mozilla/5.0 (compatible; MSIE 9.0)",
  "ip": "192.168.24.44",
  "request": "/index.html"
  "response": {
    "status": 200,
    "bytes": 52353
  },
  "ua": { "os": "Windows 7" }
}
```

在 Logstash 中调用 sprintf 来处理日志。可以在其他字符串中引用字段值。例如，在 statsd 输出有一个增量的设置，可以通过状态码来保存 apache 日志的计数：

```
output {
  statsd {increment => "apache.%{[response][status]}"}
}
```

可以使用“+ 格式”的语法来格式化时间格式。例如，如果使用基于小时和类型字段的文件输出日志：

```
output {
  file {path => "/var/log/%{type}.%{+YYYY.MM.dd.HH}"}
}
```

3. 条件语法

在 Logstash 中可以用条件语法来支持不同条件做不同的处理，这种方式和其他开发语言基本一致，例如：

```
if EXPRESSION {
```

```

...
} else if EXPRESSION {
...
} else {
...
}

```

用在过滤器中:

```

filter {
  if [action] == "login" {mutate { remove_field => "secret" }}
}

```

用在输出中:

```

output {
  # Send production errors to pagerduty
  if [loglevel] == "ERROR" and [deployment] == "production" {
    pagerduty {
      ...
    }
  }
}

```

4. 示例

我们取一条典型的 nxlog 日志, 保存文件名为 nxlog_access.log:

```

117.89.20.205 - - [03/Nov/2015:09:20:10 +0800] "POST /api/switch/list
HTTP/1.1" 200 304 "http://secilog.secisland.com/api/" "Mozilla/5.0 (Windows
NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95
Safari/537.36"

```

编写配置文件, 文件名为 logstash_nxlog.conf:

```

input {
  file {
    path => "D:/work/es_book/logstash-2.3.2/log/nxlog_access.log.txt"
    start_position => beginning
  }
}
filter {
  grok {
    match => { "message" => "%{COMBINEDAPACHELOG}" }
  }
}
output {
  elasticsearch { hosts => ["127.0.0.1:9200"] }
  stdout { codec => rubydebug }
}

```

启动 Elasticsearch。

运行 logstash -f logstash_nxlog.conf, 可以得出如下输出:


```

{
  "message" => "117.89.20.205 - - [03/Nov/2015:09:20:10 +0800] \"POST /api/switch/list HTTP/1.1\" 200 304 \"http://secilog.secisland.com/api/\" \"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36\"\\r\\n",
  "@version" => "1",
  "@timestamp" => "2016-05-21T08:20:57.970Z",
  "path" => "D:/work/es_book/logstash-2.3.2/log/nxlog_access.log",
  "host" => "seci-zhulin",
  "clientip" => "117.89.20.205",
  "ident" => "-",
  "auth" => "-",
  "timestamp" => "03/Nov/2015:09:20:10 +0800",
  "verb" => "POST",
  "request" => "/api/switch/list",
  "httpversion" => "1.1",
  "response" => "200",
  "bytes" => "304",
  "referrer" => "\"http://secilog.secisland.com/api/\"",
  "agent" => "\"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36\"",
  "geoip" => {
    "ip" => "117.89.20.205",
    "country_code2" => "CN",
    "country_code3" => "CHN",
    "country_name" => "China",
    "continent_code" => "AS",
    "region_name" => "04",
    "city_name" => "Nanjing",
    "latitude" => 32.0617,
    "longitude" => 118.77780000000001,
    "timezone" => "Asia/Shanghai",
    "real_region_name" => "Jiangsu",
    "location" => [
      [0] 118.77780000000001,
      [1] 32.0617
    ]
  }
}

```

10.1.2 插件管理

Logstash 有丰富的输入、过滤、输出插件。插件可以作为独立的软件包存在，插件管理器通过 bin/logstash-plugin 的脚本来管理整个插件的生命周期，通过命令行接口（CLI）调用可以安装、卸载、升级插件。

列出现有插件：

```

bin/logstash-plugin list
bin/logstash-plugin list --verbose

```

```
bin/logstash-plugin list '*namefragment*'
bin/logstash-plugin list --group output
```

增加插件:

```
bin/logstash-plugin install logstash-output-kafka
```

更新插件:

```
bin/logstash-plugin update
bin/logstash-plugin update logstash-output-kafka
```

移除插件:

```
bin/logstash-plugin uninstall logstash-output-kafka
```

1. 输入插件

在 Logstash 中内置非常多的输入插件, 最新的版本 2.3 中有 49 种插件:

beats、couchdb_changes、drupal_dblog、elasticsearch、exec、eventlog、file、ganglia、gelf、generator、graphite、github、heartbeat、heroku、http、http_poller、irc、imap、jdbc、jmx、kafka、log4j、lumberjack、meetup、pipe、puppet_factor、relp、rss、rackspace、rabbitmq、redis、salesforce、snmptrap、stdin、sqlite、s3、sqs、stomp、syslog、tcp、twitter、unix、udp、varnishlog、wmi、websocket、xmpp、zenoss、zeromq。

每种输入插件都有不同的属性, 基本的语法如下。

如果是文件类型的:

```
input {
  file {
    path => "nxlog_access.log.txt"
    start_position => beginning
  }
}
```

如果是 syslog 类型的:

```
input {
  syslog {port=> 514}
}
```

更详细的说明请参考官方文档。

2. 过滤插件

在 Logstash 中内置非常多的过滤插件, 最新的版本 2.3 中有 42 种插件:

aggregate、alter、anonymize、collate、csv、cidr、clone、cipher、checksum、date、de_dot、dns、drop、elasticsearch、extractnumbers、environmet、elapsed、fingerprint、geoip、grok、i18n、json、json_encode、kv、mutate、metrics、multiline、metaevent、prune、punct、ruby、range、syslog_pr、sleep、split、throttle、translate、uuid、urldecode、useragent、xml、zeromq。

每种过滤插件都有不同的属性，基本的语法如下。

如果是 grok 类型的：

```
filter {
  grok {
    match => { "message" => "%{IP:client} %{WORD:method} % {URIPATHPARAM:
      request} %{NUMBER:bytes} %{NUMBER:duration}" }
  }
}
```

如果是 json 类型的：

```
filter {
  json {
    add_field => {
      "foo_%{somefield}" => "Hello world, from %{host}"
      "new_field" => "new_static_value"
    }
  }
}
```

更详细的说明请参考官方文档。

3. 输出插件

在 Logstash 中内置非常多的输出插件，最新的版本 2.3 中有 56 种插件：

boundary、circonus、csv、cloudwatch、datadog、datadog_metrics、email、elasticsearch、elasticsearch_java、exec、file、google_bigquery、googl_cloud_storage、ganglia、gelf、graphtastic、graphite、hipchat、http、irc、influxdb、juggernaut、jira、kafka、lumberjack、librato、loggly、mongod、metriccather、nagios、null、nagios_nsca、opentsdb、pagerduty、pipe、riemann、redmine、rackspace、rabbitmq、redis、riak、s3、sqs、stomp、statsd、solr_http、sns、syslog、stdout、tcp、udp、webhdfs、websocket、xmpp、zabbix、zeromq。

每种输出插件都有不同的属性，基本的语法如下。

如果是 elasticsearch 类型的：

```
output {
  elasticsearch {
    hosts=>"127.0.0.1:9200","127.0.0.2:9200"
    index=>"logstash-%{+YYYY.MM.dd}"
  }
}
```

如果是 syslog 类型的：

```
output {
  syslog {facility => ...host => ...port => ...severity => ...}
}
```

更详细的说明请参考官方文档。

4. 编解码插件

编解码插件主要是为了对事件进行处理改变事件的数据内容。编解码插件是一个基本的过滤器，可以作为一个输入或输出的一部分。

在 Logstash 中内置非常多的编码插件，最新的版本 2.3 中有 25 种插件：

avro、cef、compress_spooler、cloudtrail、cloudfront、collectd、dots、edn_lines、edn、es_bulk、fluent、gzip_lines、graphite、json_lines、json、line、msgpack、multiline、netflow、nmap、oldlogstashjson、plain、rubydebug、s3_plain、spool。

每种编解码插件都有不同的属性，基本的语法如下。

如果是 avro 类型的：

```
input {
  kafka {
    codec => {
      avro => {schema_uri => "/tmp/schema.avsc"}
    }
  }
}
```

如果是 codec 类型的：

```
output {
  kafka {
    codec => plain {
      format => "%{message}"
    }
  }
}
```

更详细的说明请参考官方文档。

10.2 Kibana 配置

Kibana 安装也是比较简单的，也是绿色的，解压后直接运行即可。默认端口为 5601。启动后在浏览器输入：<http://127.0.0.1:5601/>。

第一次需要进行配置，在 Index name or pattern 里面填写需要搜索的索引，如图 10-2 所示。可以用通配符，选择时间字段（Time-field name），点击 Create 进入下一步的配置。

进入下一步后，可以手动调整映射，可以设置 Kibana 的默认索引，还可以在左边的 Add New 栏中添加配置其他索引，如图 10-3 所示。

完成上面的配置后，选择最上面的 Discover 就会显示查询日志主界面，如图 10-4 所示。

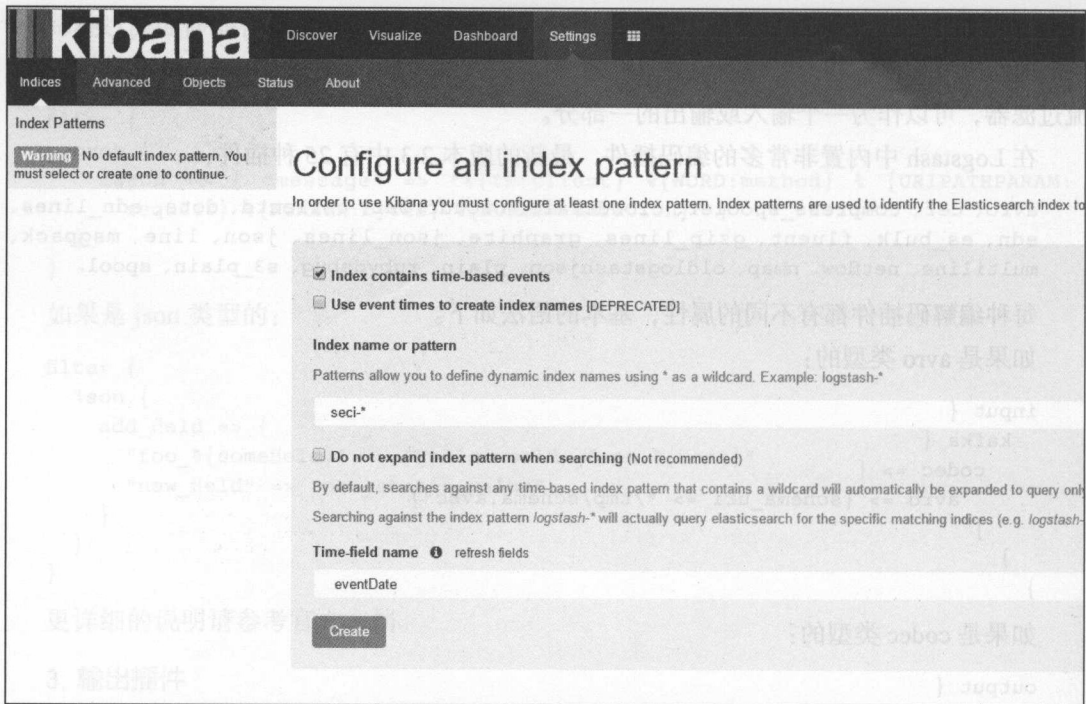


图 10-2 Kibana 配置：填写索引

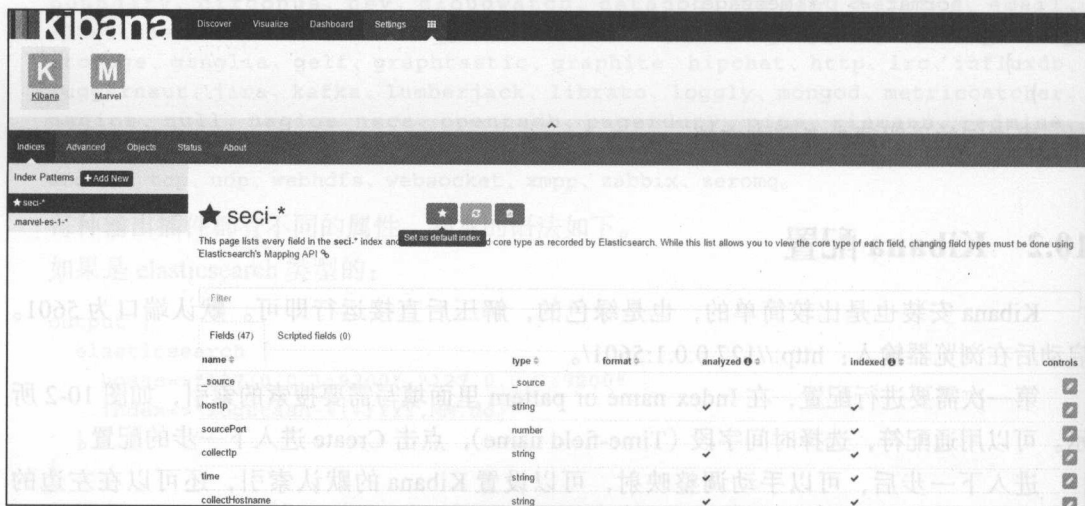


图 10-3 Kibana 配置界面：调整映射

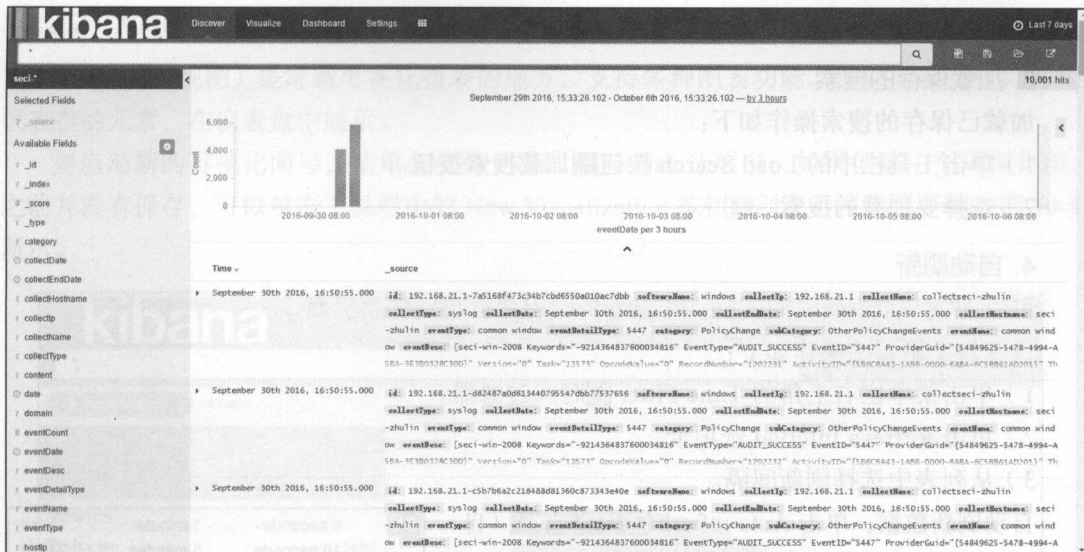


图 10-4 Kibana 查询日志图

10.2.1 Discover

点击图 10-4 最上面的 Discover (搜索) 菜单就可以查询 Logstash 存储到 Elasticsearch 中的日志。

可以通过 Discover 菜单交互地从索引中搜索数据。可以访问搜索到的每一个索引中的每一个文档。可以提交搜索查询, 筛选搜索结果和查看文档数据。还可以看到匹配搜索查询和获取字段值统计的文档的数量。如果将一个时间字段配置为所选择的搜索模式, 则文档的分布随着时间的推移显示在页面顶部的直方图中。

1. 开始一个新的搜索

在最上面的输入框中清空以前的内容, 输入新的搜索内容, 然后在搜索工具条中点击 New Search 按钮, 如图 10-5 所示。

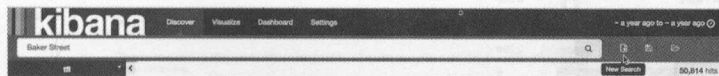


图 10-5 Kibana 空的查询日志图

2. 保存搜索

可以对之前的搜索条件进行保存, 保存后方便重新加载搜索页面, 并使用它们作为可视化的查询。


保存当前搜索步骤如下:

1) 单击工具栏中的 Save Search 按钮来保存。

2) 输入搜索的名称, 然后单击 Save 按钮。

3. 加载保存的搜索


加载已保存的搜索操作如下:


- 1) 单击工具栏中的 Load Search 按钮加载搜索按钮。
- 2) 选择要加载的搜索。

4. 自动刷新



可以配置一个刷新时间间隔, 用最新的索引数据自动刷新页面。

设置自动刷新的操作如下:

- 1) 单击菜单栏右上角的 Time Filter 时间过滤器。
- 2) 单击 Refresh Interval 选项卡。
- 3) 从列表中选择刷新间隔。

自动刷新数据, 单击 Auto-refresh  Auto-refresh (自
动刷新) 按钮并选择自动刷新的时间间隔, 如图 10-6
所示。

Off	5 seconds	1 minute	1 hour
	10 seconds	5 minutes	2 hour
	30 seconds	15 minutes	12 hour
	45 seconds	30 minutes	1 day

当自动刷新功能可用时, Kibana 的顶栏显示一个  图 10-6 Kibana 自动刷新选择图
暂停按钮和自动刷新闻隔:  1 hour。单击 Pause 按钮暂停自动刷新。

5. 查看字段数据统计

从图 10-7 左边的字段列表 (Available Fields) 中, 可以看到文档表中的字段, 点击字段
会显示这个字段排行前 5 的内容和占总数的比例。点击字段右边的 add 按钮, 会把字段添加
到主界面的列表中, 点击字段排行下面的 Visualize 按钮, 可以链接到 Visualize 页面。

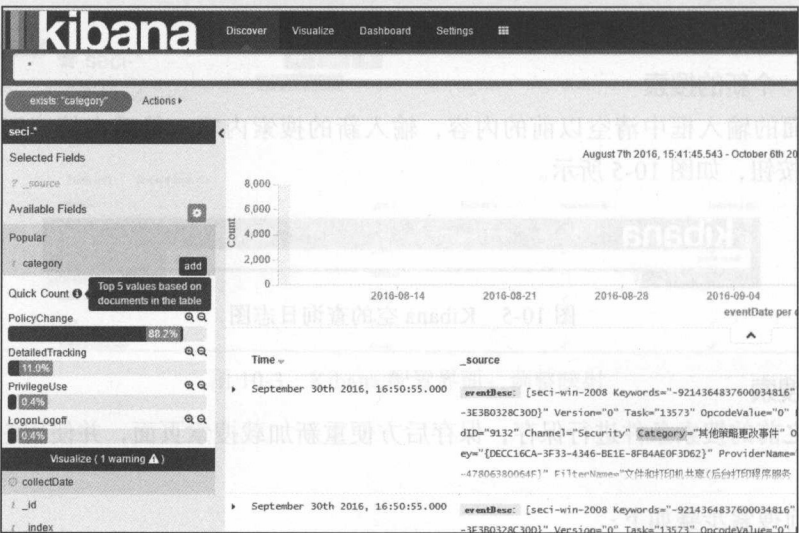



图 10-7 Kibana 数据统计图

10.2.2 Visualize

Visualize（视图）是定制可视化报表的地方，支持多种图表功能。这些图表还可以作为仪表盘的元素，在仪表盘中展示。

要启动新的可视化向导，请单击页面上端的 Visualize 选项卡。如果创建一个可视化图，之前并没有保存，可以单击工具栏中的 New Visualization 按钮创建可视化图。例如图 10-8 所示。

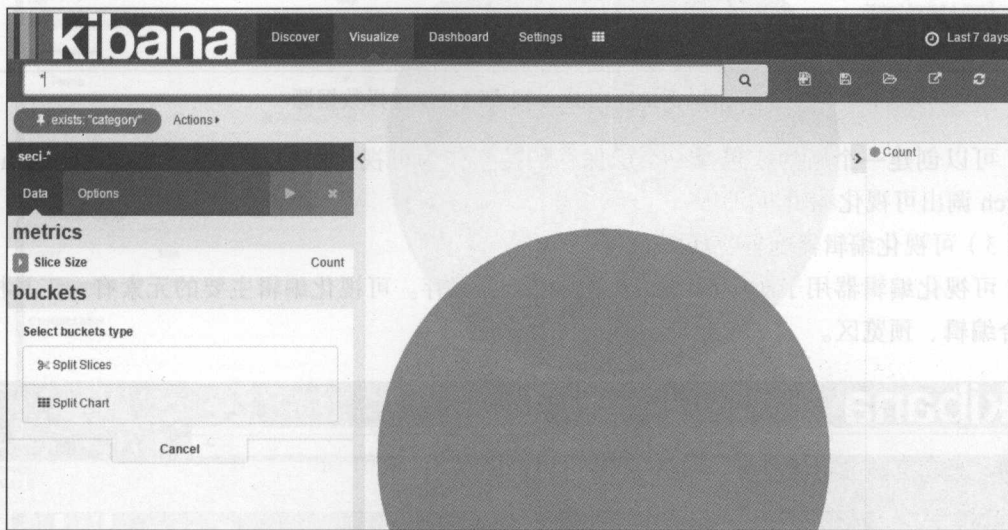


图 10-8 Kibana 未保存的 Visualize

创建可视化图会以向导的方式进行引导操作：

1) 选择一个图表类型，如图 10-9 所示。

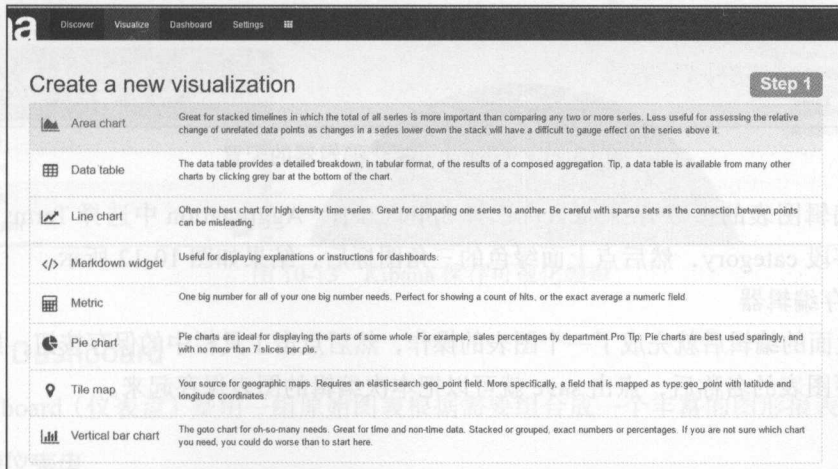


图 10-9 Kibana 报表第一步选择图表类型

2) 选择一个数据源, 如图 10-10 所示。

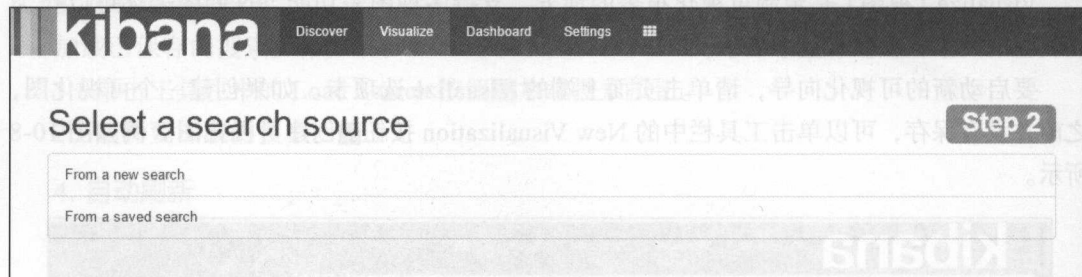


图 10-10 Kibana 报表第二步选择数据源

可以创建一个新的或选择一个已保存的搜索作为可视化图的源数据。点击 From a new search 调出可视化编辑器。

3) 可视化编辑器配置, 如图 10-11 所示。

可视化编辑器用于对所选的图表进行编辑并保存。可视化编辑主要的元素有: 工具栏、聚合编辑、预览区。

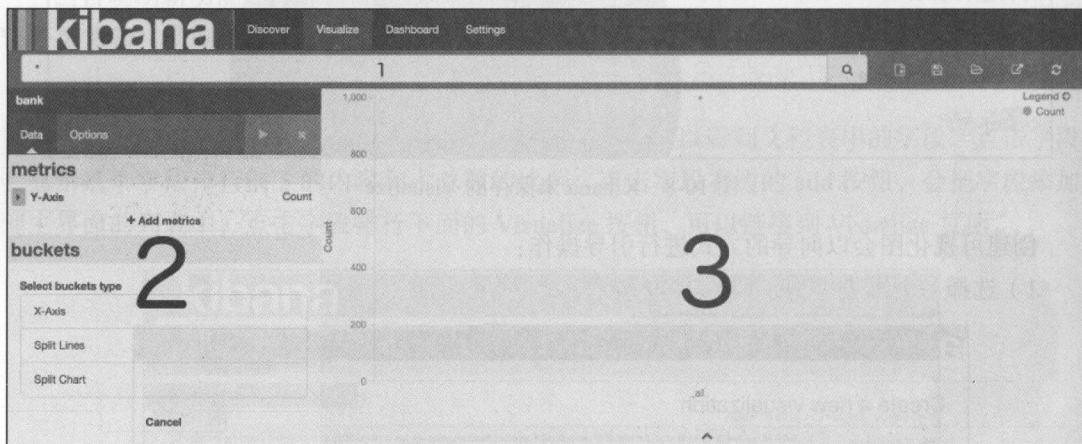


图 10-11 Kibana 可视化编辑器的配置

本例编辑图表的参数有: buckets 选择 Split Chart, Aggregation 中选择 Terms, Field 选择具体的字段 category, 然后点上面绿色的三角图标后, 结果如图 10-12 所示。

4) 保存编辑器

经过上面的编辑后就完成了一个图表的操作, 然后点击工具栏中的保存按钮, 如图 10-13 所示。填写图表的名称后, 点击 save 就可以把本次编辑的图表保存起来。

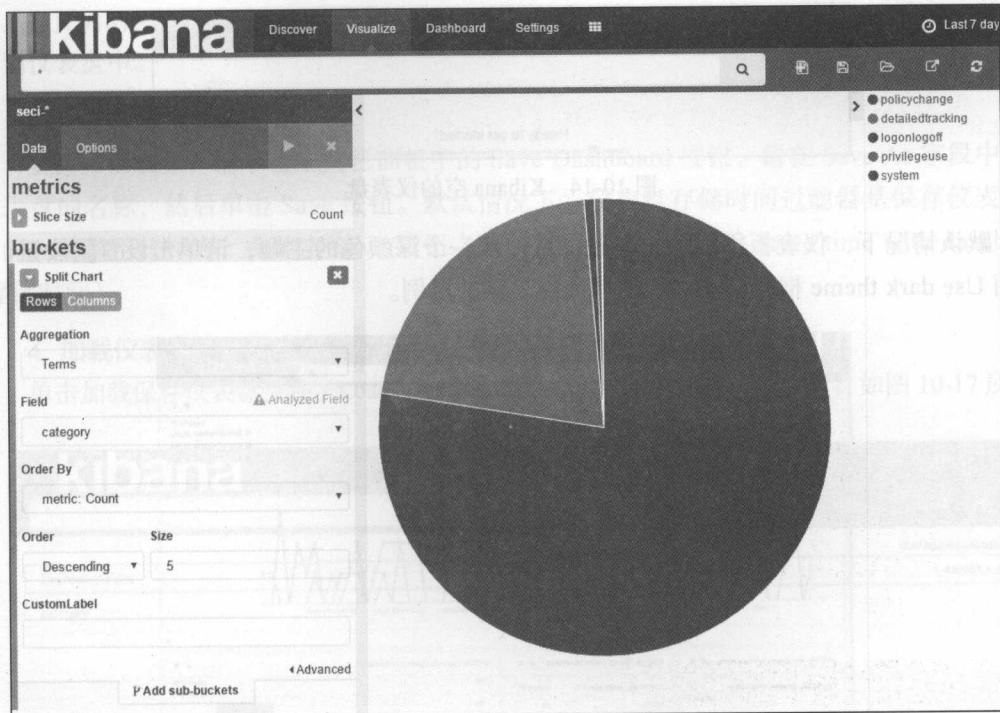


图 10-12 Kibana 可视化编辑的效果

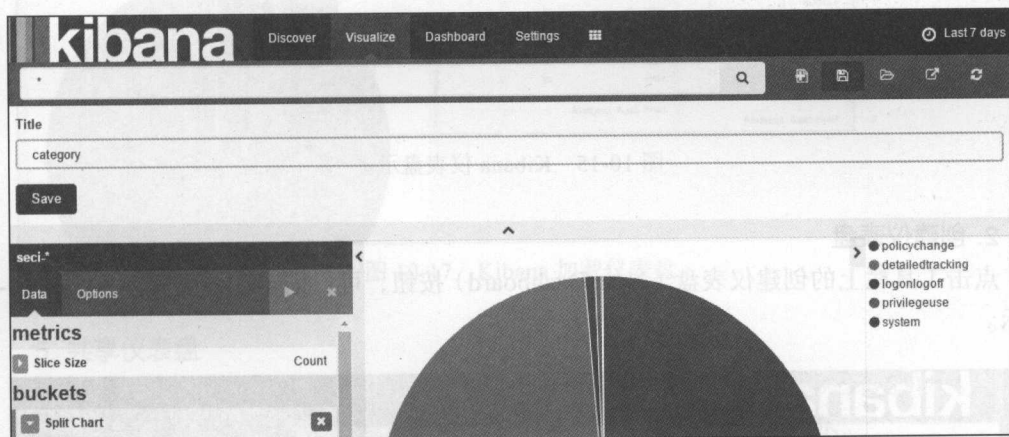


图 10-13 Kibana 保存可视化编辑

10.2.3 Dashboard

Dashboard (仪表盘) 是用一组原始图表根据需要组合成一个丰富的图形报表。

1. 空仪表盘

第一次单击 Dashboard 选项卡的时候将显示一个空的仪表盘, 如图 10-14 所示。

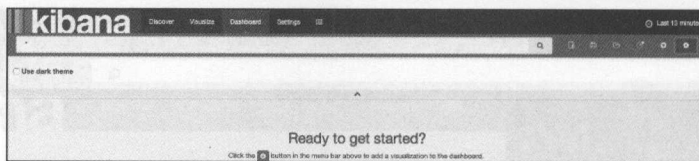


图 10-14 Kibana 空的仪表盘

默认情况下，仪表板用浅颜色主题。要使用一个深颜色的主题，请单击设置按钮，并使用 Use dark theme 框。图 10-15 是一个仪表盘的示例。



图 10-15 Kibana 仪表盘示例

2. 创建仪表盘

点击工具栏上的创建仪表盘（New Dashboard）按钮，可以创建新的仪表盘，如图 10-16 所示。

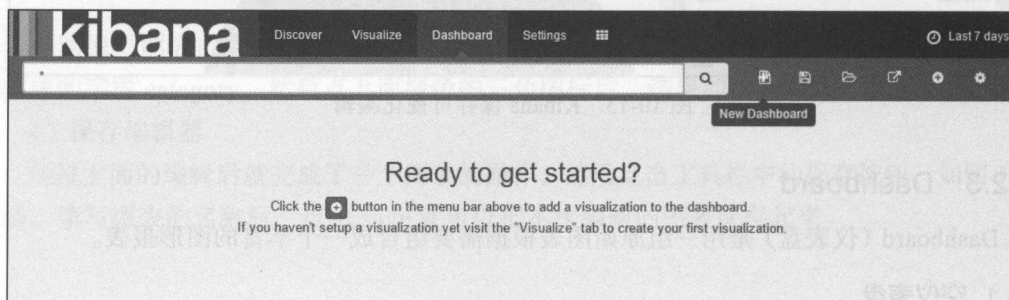



图 10-16 Kibana 创建仪表盘

在页面的最中间点击  图标，或者点击工具栏上同样的图标可以选择已经保存的视图加载到仪表盘中。

3. 保存仪表盘

要保存仪表盘，请单击工具栏面板中的 Save Dashboard 按钮，请在 Save As 字段中输入仪表盘的名称，然后单击 Save 按钮。默认情况下，仪表盘存储时间过滤器是保存仪表盘的时间。要禁用此行为，在单击 Save 按钮之前，要用仪表盘框清除 Store time with dashboard (存储时间)。

4. 加载仪表盘

单击加载保存仪表盘按钮 (Load SavedDashboard) 将显示现有的仪表盘，如图 10-17 所示。

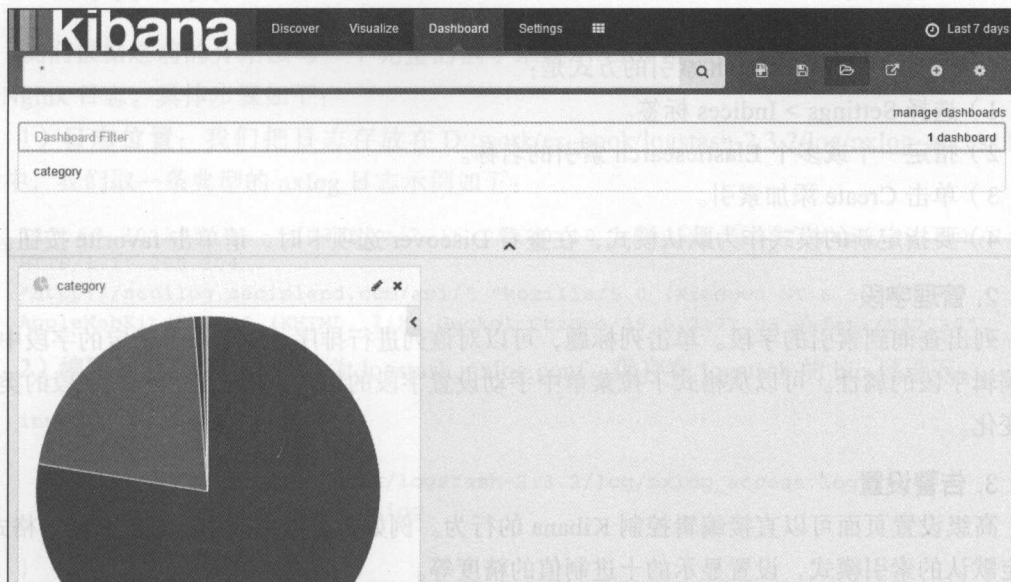



图 10-17 Kibana 加载仪表盘

5. 共享仪表盘

可以和其他用户共享仪表盘，分享一个链接到仪表盘或将仪表盘嵌入到网页中。要分享一个仪表盘，点击分享按钮  显示共享面板，如图 10-18 所示。

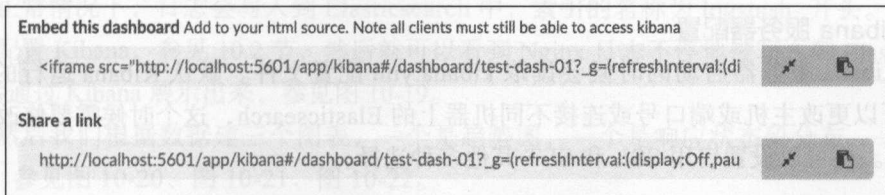


图 10-18 Kibana 共享仪表盘

点击 Copy to Clipboard (复制按钮) 复制本地 URL 或嵌入的 HTML 到剪贴板。点击 Generate short URL (生成短网址) 的按钮 创建短网址分享或嵌入的链接。

10.2.4 Settings

Kibana 和配置相关的地方都在 Settings 菜单中, 用这个菜单可以设置 Elasticsearch 的索引, 管理索引的字段, 设置一些高级参数, 设置 Kibana 服务器的参数, 管理之前介绍的搜索、视图和仪表盘等。

1. 索引设置

可以设置 Kibana 识别的索引模式, 支持一个或一个以上的 Elasticsearch 索引。可以用星号 (*) 匹配零个或多个索引。例如, myindex * 匹配所有名字以 myindex 开始的索引, 如 myindex-1 和 myindex-2。索引识别也可以简单的是一个索引的名称。

创建连接到 Elasticsearch 索引的方式是:

- 1) 选择 Settings > Indices 标签。
- 2) 指定一个或多个 Elasticsearch 索引的名称。
- 3) 单击 Create 添加索引。
- 4) 要指定新的模式作为默认模式, 在查看 Discover 选项卡时, 请单击 favorite 按钮。

2. 管理字段

列出查询到索引的字段。单击列标题, 可以对该列进行排序。对于一个给定的字段中单击编辑字段的属性。可以从格式下拉菜单中手动设置字段的格式。格式选项根据字段的类型而变化。

3. 告警设置

高级设置页面可以直接编辑控制 Kibana 的行为。例如, 可以更改用于显示日期的格式, 指定默认的索引模式, 设置显示的十进制值的精度等。

设置高级选项:

- 1) 选择 Settings → Advanced 标签。
- 2) 单击要修改的选项的 Edit 按钮。
- 3) 为选项输入一个新值。
- 4) 单击 Save 按钮。

4. Kibana 服务器配置

在 Kibana 服务器启动的时候会读取 kibana.yml 配置文件。默认 Kibana 运行的端口是 5601。可以更改主机或端口号或连接不同机器上的 Elasticsearch, 这个时候需要修改 kibana.yml 文件。还可以设置启用 SSL 和 SET 其他多种选择。

5. 管理搜索，可视化和仪表板

可以从 Settings → Objects 中查看、编辑和删除已保存的搜索、可视化、仪表板。还可以导出或导入的搜索、可视化和仪表板。

- 1) 转到 Settings → Objects 标签。
- 2) 选择要查看的对象。
- 3) 单击“查看”按钮。

编辑保存的对象，可以直接修改对象。可以改变对象的名称，添加描述，定义和修改对象的属性的 JSON 字符串。

10.3 综合示例

我们根据之前的介绍编写一个完整的例子来展示 ELK 的强大。日志源为我们比较常见的 Nginx 日志。具体步骤如下：

1) 日志位置：我们把日志存放在 D:/work/es_book/logstash-2.3.2/log/nxlog_access.log.txt 中，我们取一条典型的 nxlog 日志示例如下：

```
117.89.20.205 - - [03/Nov/2015:09:20:10 +0800] "POST /api/switch/list
HTTP/1.1" 200 304
"http://secilog.secisland.com/api/" "Mozilla/5.0 (Windows NT 6.3; WOW64)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/39.0.2171.95 Safari/537.36"
```

2) 编写配置文件，文件名为 logstash_nxlog.conf，保存在 logstash 的 bin 目录下：

```
input {
  file {
    path => "D:/work/es_book/logstash-2.3.2/log/nxlog_access.log.txt"
    start_position => beginning
  }
}
filter {
  grok {match => { "message" => "%{COMBINEDAPACHELOG}"}}
}
output {elasticsearch { hosts => ["127.0.0.1:9200"] }}
```

3) 启动 Elasticsearch。

4) 运行 logstash -f logstash_nxlog.conf。

5) 正常情况下，日志会写入到 Elasticsearch 中，索引的名称为 logstash- 开头。

6) 配置 Kibana，参见 10.2 节，然后就可以看到 Nginx 日志不停地被写入到 Elasticsearch 中，然后通过 Kibana 展示出来，参见图 10-19。

7) 然后我们根据数据建三个图表，一个是趋势图，一个是响应状态码分布，一个是源 IP 分布，参见图 10-20、图 10-21、图 10-22。

图 10-22 源 IP 分布

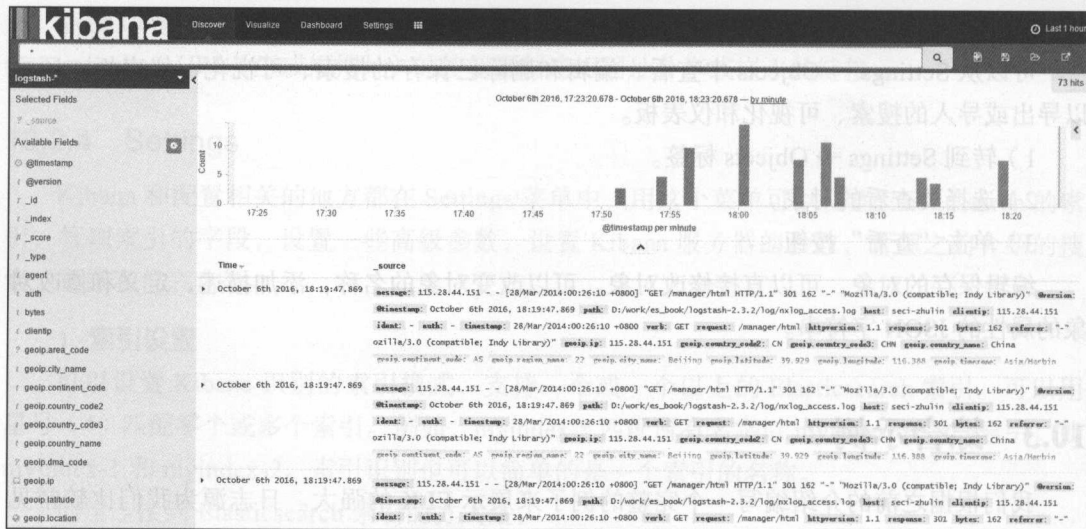


图 10-19 Kibana 加载仪表盘

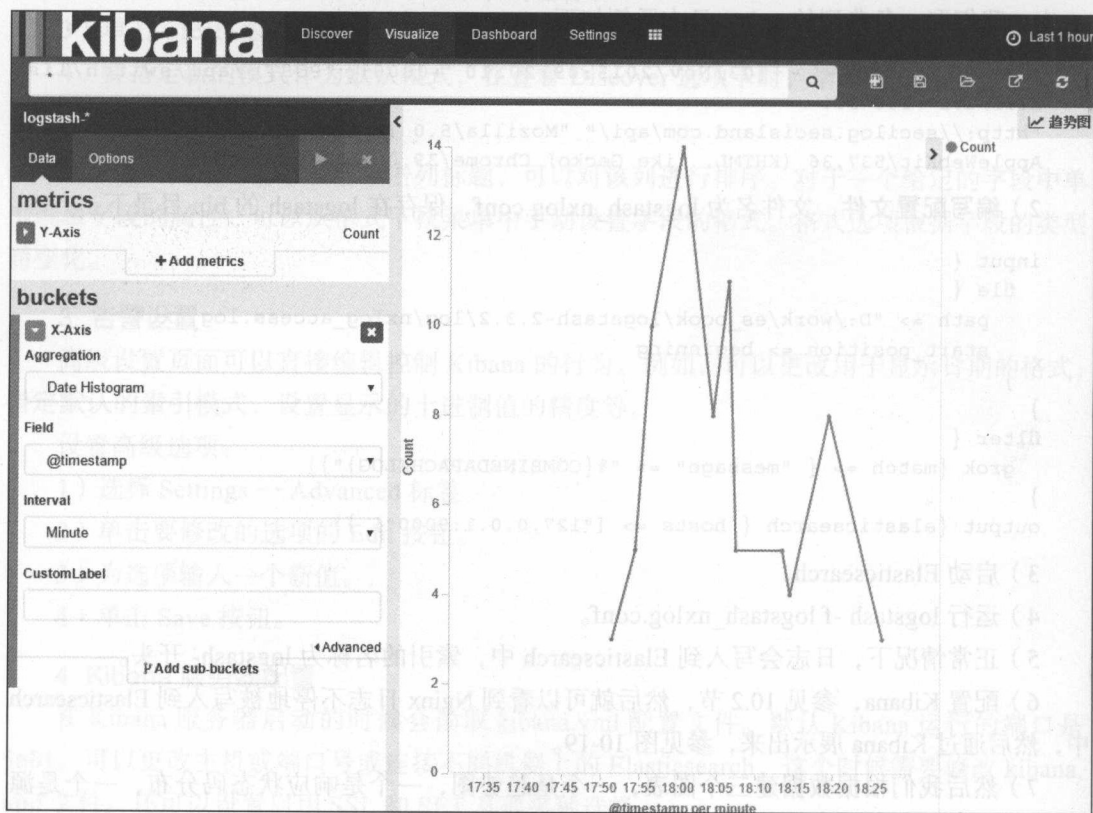


图 10-20 事件趋势图

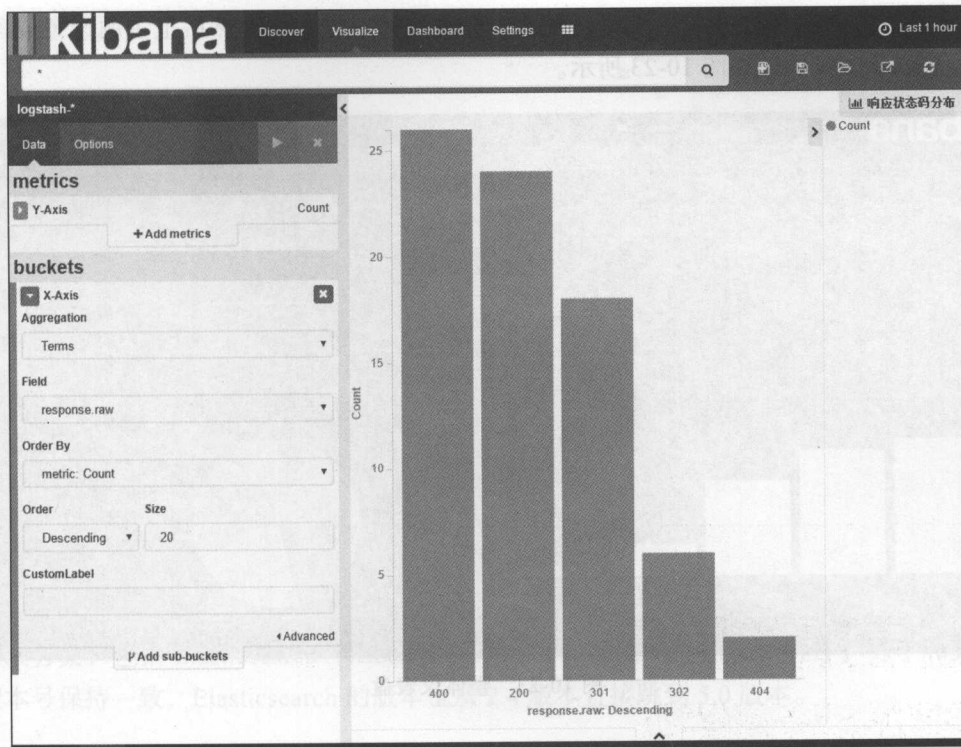


图 10-21 响应状态码分布

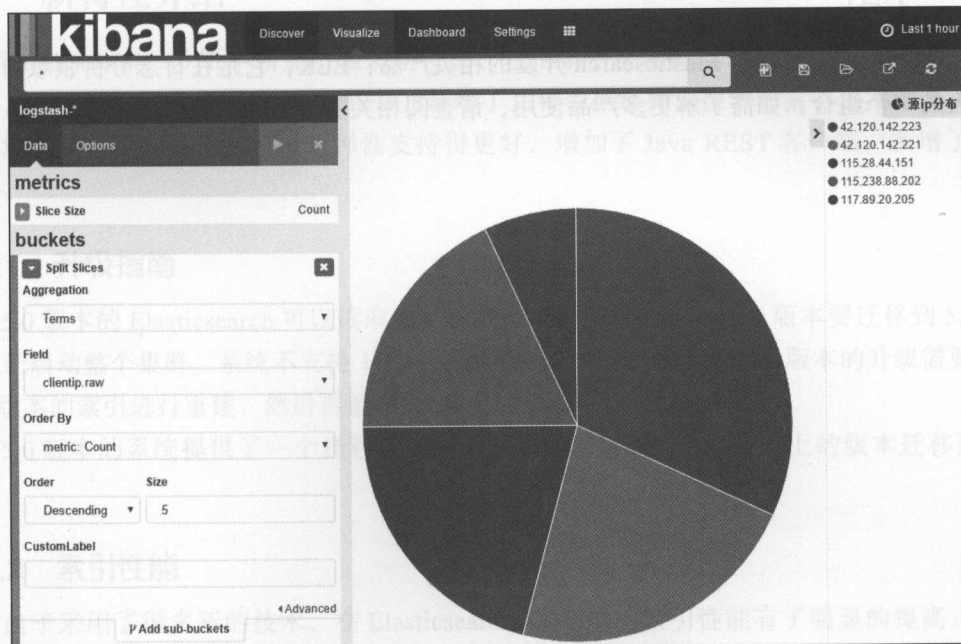


图 10-22 源 IP 分布

8) 最后, 我们建立一个仪表盘, 把三个图表放到仪表盘中, 然后把主题设置成深色, 这样看起来比较专业, 如图 10-23 所示。

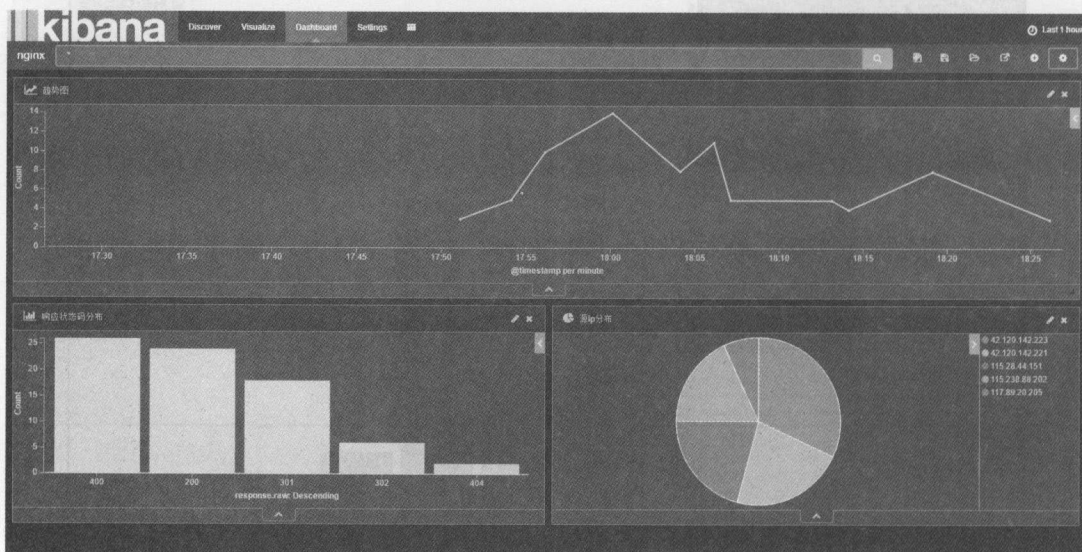


图 10-23 整体仪表盘

10.4 小结

本章主要介绍了基于 Elasticsearch 开发的相关产品: ELK, 它是在日志分析领域内目前非常火的一个组合, 如需了解更多产品使用, 请查阅相关资料。

Elasticsearch 5.0 的特性与改进

2016 年 10 月 27 日 Elastic 公司发布了全新的 ElasticStack 5.0.0 版本，Elastic 公司有一套解决方案，包括 Elasticsearch、Kibana、Beats、Logstash 几个主要的程序。为了和其他产品版本号保持一致，Elasticsearch 的版本也从 2.4 版本直接跳到 5.0 版本。

A.1 新特性介绍

Elasticsearch 5.0 版本增加了很多新特性，包括索引性能的提升，新增了数据转换节点类型，新增了一种内置的脚本 Painless，新增了几种数据结构，搜索和索引也发生了一些变化，增加了几个友好性接口，对弹性支持得更好，增加了 Java REST 客户端，新增了迁移助手。

A.1.1 升级指南

5.0 版本的 Elasticsearch 可以读取 2.X 版本生成的索引，所以 2.X 版本要迁移到 5.X，需要重新启动整个集群。系统不支持 1.X 版本到 5.X 版本的升级。对 1.X 版本的升级需要先对 1.X 版本的索引进行重建，然后再把索引加到 5.0 版本中。

5.0 版本的系统提供了一个迁移助手插件，可以帮助系统从 2.3 以上的版本迁移到 5.X 版本。

A.1.2 索引性能

由于采用了很多新的技术，在 Elasticsearch 5.0.0 中的索引性能有了明显的提高。这些新技术包括采用了新的数字类型结构，在并发更新到同一文档中采用了新的技术减少了锁

的冲突，在同步事务日志时减少锁定的条件。采用了异步同步日志机制，减少了磁盘寻址的时间。Elasticsearch 自动生成文档 ID 时采用了基于时间的用户实例，增加了比较大的效率。支持实时获取文档内部的更改，可以增加更多的可用内存用于索引缓冲区，同时在垃圾集中花费更少的时间。通常情况，可以提高 25% ~ 80% 的效率。

A.1.3 数据转换节点

在 Elasticsearch 5.0.0 中，新增了数据转换节点 (Ingest Nodes)。它可以不依赖于 Logstash 实现常用的过滤能力，比如 `grok`、`split`、`convert`、`date` 等。它可以用来执行常见的数据转换和处理。可以使用转换节点在实际索引之前对文档进行预处理。在任何转换节点中，处理索引或者块处理之前进行预处理转换。可以在任何节点开启转换功能，或者建立单独的转换节点。在默认情况下，在任何节点都开启了转换能力，如果要关闭转换能力，需要在配置文件中添加：`node.ingest: false`。

在索引文档之前进行预处理，它定义了一个指向一系列处理器的管道。每个处理器以某种方式转换文档。例如，你可能有一个管道，包括一个处理器，从文档中删除字段，然后进入另一个处理器，对文档中的字段进行重命名。

使用一个管道，你只需在一个索引或批量请求后加入管道参数。例如：

请求：PUT `http://127.0.0.1:9200/secisland/secilog/my-id?pipeline=my_pipeline_id`

参数：

```
{
  "foo": "bar"
}
```

在使用前，需要先定义管道，例如定义上面的管道：

请求：PUT `http://127.0.0.1:9200/_ingest/pipeline/my-pipeline-id`

参数：

```
{
  "description" : "describe pipeline",
  "processors" : [
    {
      "set" : { "field": "foo", "value": "bar" }
    }
  ]
}
```

A.1.4 新增脚本语言

由于外部的脚本引擎太过于强大，什么都能做，用不好或者设置不当就会引起安全风险，基于安全和性能方面的考虑，在 Elasticsearch 5.0 中引入了一个新的脚本引擎 Painless，顾名思义，简单安全，无痛使用，这个脚本引擎默认是开启的。和 Groove 的沙盒机制不一

样, Painless 使用白名单来限制函数与字段的访问, 针对 Elasticsearch 的场景来进行优化, 只做 Elasticsearch 数据的操作, 更加轻量级, 速度要快好几倍, 并且支持 Java 静态类型, 语法保持和 Groove 类似, 还支持 Java 的 lambda 表达式。

A.1.5 新的数据结构

在 Elasticsearch 5.0 版本中集成了 Lucene 6, Lucene 6 对数字类型和地理类型增加了一个新的点数据结构, 叫做 K-Ds 树, 它改变了数字类型的索引和搜索。增加了 36% 的查询速度, 增加了 71% 索引速度, 减少了 66% 的硬盘空间占用, 减少了 85% 的内存使用。新增加的 IP 字段支持 IP4 和 IP6。根据 Lucene 6 新的 LatLonPoint 结构修改了地理点类型, 使地理点查询的性能增加了一倍。同时还增加了一个半精度 (half_float) 浮点类型 (16 位) 和大浮点类型 (scaled_float), 它们使用了数据压缩技术来减少磁盘占用。这些新类型意味着在许多情况下, 特别是在数字型数据中将显著减少磁盘占用的空间。

A.1.6 友好性接口变化

1. 索引的改进

由于 Elasticsearch 中的配置实在太多, 在新的 5.0 版本中, 对配置验证更加严格, 保证原子性, 如果其中一项失败, 那整个更新请求都会失败, 不会一半成功一半失败。新增了一个 Shrink 接口

在 Elasticsearch 5.0 之前的版本中, 索引的分片数是固定的, 设置好了之后不能修改, 如果发现分片太多或者太少, 如果要修改, 只能重建索引。在 5.0 中增加了 Shrink 接口, 它可将分片数进行收缩成它的因数。如之前是 15 个分片, 你可以收缩成 5 个或者 3 个又或者 1 个, 那么我们就可以想象成这样一种场景, 在写入压力非常大的收集阶段, 设置足够多的索引, 充分利用 shard 的并行写能力, 索引写完之后收缩成更少的 shard, 提高查询性能。新增了一个 Rollover 接口

对于日志类的数据一般按天来对索引进行分割 (数据量更大还能进一步拆分)。以前是在程序里设置一个自动生成索引的模板, 例如 logstash 中的 logstash-[YYYY-MM-DD] 模板, 现在 5.0 版中提供了一个更加简单的方式: Rollover API, 例如:

请求: PUT http://127.0.0.1:9200/logs-000001

参数:

```
{
  "aliases": {
    "logs_write": {}
  }
}
```

Add > 1000 documents to logs-000001

请求: POST http://127.0.0.1:9200/logs_write/_rollover

参数:

```
{
  "conditions": {
    "max_age": "7d",
    "max_docs": 1000
  }
}
```

从上面可以看到, 首先创建一个 logs-0001 索引, 它有一个别名是 logs_write, 然后给这个 logs_write 创建了一个 rollover 规则, 即这个索引文档不超过 1000 个或者最多保存 7 天的数据, 超过会自动切换别名到 logs-0002。也可以设置索引的 setting、mapping 等参数, 剩下的 Elasticsearch 会自动帮你处理。

2. 新增 Depreated logging

大家在用 Elasticsearch 的时候, 其实有些接口可能已经打上了 Depreated 标签, 即废弃了, 在将来的某个版本中就会移除, 当前能用是因为一般废弃的接口都不会立即移除, 给足够的时间迁移, 但是也是需要知道哪些不能用了, 要改应用代码了。所以现在有了 Depreated 日志, 当打开这个日志之后, 调用的接口如果已经是废弃的接口, 就会记录下日志, 那么接下来的事情你就知道应该怎么做。

3. 新增 Cluster allocation explain 接口

有人问: “谁能给我一个 shard 不能分配的理由?”, 现在有了, 如果之前遇到过分片不能正常分配的问题, 但是不知道是什么原因, 只能尝试手动路由或者重启节点, 但是不一定能解决, 其实里面有很多原因, 现在提供的这个 explain 接口就是告诉你目前为什么不能正常分配的原因, 方便解决问题。

A.1.7 弹性

在 Elasticsearch 5.0 版本中, 分布式模型的每一部分都已分离, 重构, 简化, 并使之更可靠。群集状态更新要等待集群中的所有节点确认。当主分片复制失败的时候, 需要等待 master 节点的响应。索引的数据路径现在使用 UUID, 而不是索引名称, 这样可以避免命名冲突。

在 Elasticsearch 5.0 中增加了启动检查来确保配置的正确性。同时引入了开发模式和生产模式两种方式, 对生产模式的检查会更加严格。系统增加了新的内存控制器, 在聚合请求等过程中监控内存的使用, 当聚合内存有可能过大的时候则中断查询请求, 正常情况下, 内存不足的情况基本上不会发生了。

对多用户的情况下, 系统增加了更多的权限控制来减少普通用户的误操作, 例如: 请求超时后禁用文本字段的数据加载, 限制分片请求的数量, 限制映射的字段数量等。

A.1.8 Java RestClient 客户端

在 Elasticsearch 5.0 中提供了一个 Java 原生的 REST 客户端 SDK，相比之前的 TransportClient，版本依然简单，集群升级不影响，支持跨 Java 版本的调用等，新的基于 HTTP 协议的客户端对 Elasticsearch 的依赖解耦，没有 jar 包冲突，提供了集群节点自动发现、日志处理、节点请求失败自动进行请求轮询，充分发挥 Elasticsearch 的高可用能力，并且性能不相上下。

A.1.9 迁移助手

在 Elasticsearch 5.0 中提供了一个迁移助手，它是一个网站的插件，可以帮助你从 Elasticsearch 2.3.x/2.4.x 版本到 5.x 版本的迁移。配备了三个工具：

- 集群的检查：在群集、节点和索引上运行一系列检查，并提醒在升级之前需要纠正的任何已知问题。
- 重建索引助手：在 V2.0.0 版本之前的索引需要重建才可以在 5.0 中识别。重建索引助手通过一个按钮升级索引。
- 过期接口日志：当使用过期接口时，系统会把过期的日志打出来，助手可以在集群中打开或者关闭过期日志打印功能。

A.2 索引相关的变化

A.2.1 搜索和 DSL 查询的变化

1. search_type=count 参数移除

这个是在 2.0 版本中已经不推荐使用的参数，取而代之的是用查询，size 设为 0。

请求：POST http://127.0.0.1:9200/secisland/_search

参数：

```
{
  "size": 0,
  "aggs": {
    "my_terms": {
      "terms": { "field": "foo" }
    }
  }
}
```

2. search_type=scan 参数移除

这个是在 2.1 版本中已经不推荐使用的参数。现在滚动请求用 _doc 排序达到同样的效果。

请求: POST http://127.0.0.1:9200/secisland/_search?scroll=2m

参数:

```
{
  "sort": ["_doc"]
}
```

3. 分片查询限制

在 1000 个以内, 因为分片查询会消耗大量的内存和 CPU, 如果想要绕开这个限制, 需要在配置文件中修改参数: `action.search.shard_count.limit`。

3.1. `fields` 参数将被 `stored_fields` 参数替换, 它只返回存储的字段, 将不再从 `_source` 字段中获取。

3.2. `fielddata_fields` 参数将被 `docvalue_fields` 参数替换。

3.3. 是否存在接口取消, 取而代之的是用查询获得, 参数 `size` 设为 0, `terminate_after` 设置为 1。

4. 查询变化

4.1. 不支持 `geo_point` 的 `term` 查询。

4.2. `fuzzy` 查询在 `numeric`, `date`, `ip` 字段中失效, 取而代之的是范围查询。

4.3. 在 `_uid` 和 `_id` 字段中查询 `range` 和 `prefix` 将不被支持。

4.4. 查询不存在的索引将返回失败, 而不是返回 `no hits`。

4.5. 在 `fuzzy` 查询中将不支持 `min_similarity` 参数。

4.6. 在 `query_string` 查询中将不支持 `fuzzy_min_sim` 参数。

4.7. 在 `completion suggester` 中取消 `edit_distance` 参数。

4.8. 在索引查询中将不再支持 `no_match_filter` 参数。

4.9. 在嵌套查询中将不再支持 `filter` 字段查询。

4.10. 在 `term` 查询中将不再支持 `minimum_should_match` 和 `disable_coord`, 取消了 `execution` 参数。

4.11. 在 `_score` 查询中取消了 `top level` 过滤。

4.12. 在 `span_near` 查询中取消了 `collect_payloads` 参数。

4.13. 在 `nested` 和 `has_child` 查询中取消了 `score_type` 参数, `has_parent` 查询中取消了 `score_mode` 和 `total score mode` 类型。

4.14. 在 `has_child` 查询中 `max_children` 参数在以前的版本中设置为 0 表示没有限制, 现在表示 0。

4.15. 如果 `_field_names` 字段被禁用时, `exists` 查询将返回失败。

4.16. 如果 `cross_fields`, `phrase`, `phrase_prefix` 类型中有模糊 (fuzziness) 查询, 则 `multi_`

match 查询将失败。

4.17. GeoPolygonQuery, GeoDistanceQuery, GeoBoundingBoxQuery 查询中 coerce, normalize, ignore_malformed 参数将失效, 用 validation_method 参数代替。

4.18. geo_distance_range 查询将取消, 将用 geo_distance 聚合来代替。

4.19. 在 search 接口中 top level 过滤将取消, 用 post_filter 来代替。

4.20. 多个高亮显示名称将不被支持, 唯一支持的是 plain, fvh, postings。

4.21. term vectors 接口不再支持未映射的字段的映射。DFS 参数在 term vectors API 中已经被完全删除。

4.22. reverse 参数在排序中将被取消, 在 validation_method 方法中取消 coerce 和 ignore_malformed 参数。

4.23. Top level inner hits 语法被取消, inner hits 只能存在于 nested, has_child, has_parent 查询中。

5. 查询分析器

在 profiling 查询的返回值中, query_type 改名为 type, lucene 改名为 description

6. 搜索偏好

在搜索偏好 _only_node 被删除, 可以用 _only_nodes 和指定的 node ID 来实现。_only_nodes 替换了 _only_node 的功能, _only_node 不支持多个节点。preference_shards 参数接收用 | 分开第二个参数, 例如 _shards:2,3|_primary。

搜索结果分页增加了 search_after 特性, 它有效地跳过以前返回的结果只返回下一页的数据。移动搜索现在可以并行执行。默认使用 BM25 评分算法, 替换之前的 TF/IDF 评分算法。

A.2.2 索引映射的变化

1. String 类型

引入新的字段类型 Text/Keyword 来替换 String, 即以前的 string 类型分成 text 和 keyword 两种类型。keyword 类型的数据只能完全匹配, 适合那些不需要分词的数据, 对过滤、聚合非常友好; text 当然就是全文检索需要分词的字段类型了。将类型分开的好处就是使用起来更加简单清晰, 以前需要设置 analyzer 和 index, 并且有很多都是自定义的分词器, 从名称根本看不出到底分词没有, 用起来很麻烦。另外 string 类型暂时还保留的, 但 6.0 版本会移除。

字符串映射现在默认如下:

```
{
  "type": "text",
  "fields": {
```



```

    "keyword": { "type": "keyword", "ignore_above": 256 }
  }
}

```

字符串被映射成了两个字段，一个字段进行分词，另一个字段不进行分词，可以用于聚合或者排序。

2. 数字类型

数字类型现在是一种全新的数据结构，被称为 BKD tree。相比以前的结构有更快的对比查询效率和占用更少的磁盘空间。注意，数字类型将不参与索引的评分，如果需要对数字字段进行参与评分，可以同时映射为数字类型和 keyword 类型。例如：

请求：PUT <http://127.0.0.1:9200/secisland>

参数：

```

{
  "mappings": {
    "my_type": {
      "properties": {
        "my_number": {
          "type": "long",
          "fields": {
            "keyword": { "type": "keyword" }
          }
        }
      }
    }
  }
}

```

3. geo_point 字段

和数字类型类似，Geo point 字段类型也用了 BKD tree 结构。由于这种结构从根本上进行多维空间数据的支持，所以下面字段的参数将不再支持：Geohash, geohash_prefix, geohash_precision, lat_lon。Geohashes 在接口中仍然可以使用，但它不再是用来索引地理的数据点。

4. _timestamp 和 _ttl 字段

元字段 _timestamp 和 _ttl 字段将不再被支持，对于 _timestamp 可以在文档中添加日期字段来代替或者用 ingest pipeline，例如：

请求：PUT http://127.0.0.1:9200/_ingest/pipeline/timestamp

参数：

```

{
  "description" : "Adds a timestamp field at the current time",
  "processors" : [ {
    "set" : {

```



```

    "field": "timestamp",
    "value": "{[_ingest.timestamp]}"
  }
}

```

请求: PUT http://127.0.0.1:9200/newindex/type/1?pipeline=timestamp

参数:

```

{"example": "data"}

```

请求: GET http://127.0.0.1:9200/newindex/type/1

对于 `_ttl` 可以用 `time-based` 索引或者在一个时间戳字段范围查询 (`_delete-by-query`) 任务来替换。

请求: POST http://127.0.0.1:9200/index/type/_delete_by_query

参数:

```

{
  "query": {
    "range": {
      "timestamp": {"lt": "2016-05-01"}
    }
  }
}

```

5. 索引属性

所有在用的字段类型,除了将要废弃的 `string`,索引属性只有 `true/false` 两种,用来代替之前的 `not_analyzed/no`, `string` 字段类型还是 `analyzed/not_analyzed/no`。

6. 非索引字段的文档值

在此之前,设置一个字段的属性为 `index:no` 将禁用文档的值,现在,文档的值对数字和 `boolean` 类型的值总是有效,除非 `doc_values` 的值设置为 `false`。

7. 默认的浮点类型由 float 来代替 double

在 Elasticsearch 5.0 中动态映射的浮点类型默认为 `float` 类型,而不是 `double` 类型,因为大多数情况下 `float` 类型已经够用了,而 `float` 类型将显著的减少磁盘空间的占用。

`norms`: 现在用 `boolean` 来代替对象, `norms.enabled` 被替换成了 `boolean`、`norms.loading` 参数 `eager` 将不再起作用,现在 `norms` 是基于磁盘的。

设置 `fielddata.format: doc_values` 用于隐式启用字段的文档映射中。现在隐式方式将不再起作用,需要明确地设置 `doc_values` 的值为有效或者无效。

`fielddata.filter.regex` 参数将不再支持,未来版本中将会取消。

8. 字段映射限制

在 Elasticsearch 5.0 中对索引中的字段进行了限制,最大 1000 个字段。

字段的最大深度（嵌套字段）是 20 层。

索引中嵌套字段的最大数量是有限的 50 层。

`_parent` 字段将不再索引，连接父母与孩子之间的文件不再依赖索引字段，因此从 Elasticsearch 5.0.0 起 `_parent` 字段不再索引。为了找到文档中引用的父 id，可以使用新的 `parent_id` 来进行查询。搜索返回中的得到的响应和 `hits` 仍然包括父标识。

`_source` 映射不再支持格式选项，现在只是为了兼容性保留，将来将会被取消。

核心类型不再支持对象符号（bjeect notation），它被用来提供每个文档的 `boosts`，例如：

```
{
  "value": "field_value",
  "boost": 42
}
```

9. `_all` 查询的精度

在 `_all` 上的每个字段的长度由以前的 4 个字节压缩到了一个字节，虽然这将使索引的空间效率更高，但这也意味着索引时间的计算将不太准确。

A.2.3 percolator 类型


`percolator` 字段类型将解析 json 结构到本地并存储到索引中。因此可以用 `percolate` 查询来匹配提供的文档。这种情况可以理解为正常搜索的反向，一般情况下我们索引一个文档，然后通过搜索进行查询。`percolator` 是先存储搜索，然后用文档来进行查询是否匹配搜索。

任何含有 json 对象的列可以被配置成 `percolator` 字段，例如下面的配置是映射 `percolator` 字段类型，这种类型适用于 `percolate` 查询：

```
{
  "properties": {
    "query": {"type": "percolator"}
  }
}
```

那么下面的 JSON 代码段可以被索引为一个本地查询：

```
{
  "query" : {
    "match" : {"field" : "value"}
  }
}
```

 **注意** `percolator` 查询必须是已经存在的与所使用的 `percolation` 索引相关联的映射，为了确保这些存在的字段，通过创建索引或者设置映射来增加或者更新。`percolator` 类型可以存在任何索引的任何类型中。一个索引中只能有一个 `percolator` 类型字段。

1. percolate 查询

percolate 查询可以将存储在索引中的查询进行字段匹配。例如创建两个映射的索引：

请求：PUT <http://127.0.0.1:9200/secisland>

参数：

```
{
  "mappings": {
    "doctype": {
      "properties": {
        "message": { "type": "text" }
      }
    },
    "queries": {
      "properties": {
        "query": { "type": "percolator" }
      }
    }
  }
}
```

doctype 映射在 percolator 查询中被索引到一个临时索引之前用于预处理文件的映射。

queries 映射用于索引查询文档。query 字段将产生一个 JSON 对象代表一个实际的 Elasticsearch 查询。query 字段设置的类型为 percolator 类型，此字段类型可以理解为查询 DSL 和存储这些查询，在定义了 percolate 查询时它可以用于匹配文档。

在 percolator 中添加一个查询：

请求：PUT <http://127.0.0.1:9200/secisland/queries/1?refresh>

参数：

```
{
  "query" : {
    "match" : { "message" : "bonsai tree" }
  }
}
```

在登记 percolator 的查询中匹配文档：

请求：POST http://127.0.0.1:9200/secisland/_search

参数：

```
{
  "query" : {
    "percolate" : {
      "field" : "query",
      "document_type" : "doctype",
      "document" : { "message" : "A new bonsai tree in the office" }
    }
  }
}
```

返回值:

```
{
  "took": 13,
  "timed_out": false,
  "_shards": {"total": 5, "successful": 5, "failed": 0},
  "hits": {
    "total": 1,
    "max_score": 0.5716521,
    "hits": [
      {
        "_index": "secisland",
        "_type": "queries",
        "_id": "1",
        "_score": 0.5716521,
        "_source": {
          "query": {
            "match": {"message": "bonsai tree"}
          }
        }
      }
    ]
  }
}
```

查询的参数如下:

- ❑ **field**: 定义 percolator 字段类型的字段, 必填。
- ❑ **document_type**: 映射的稳定字段, 必填。
- ❑ **document**: 需要匹配的原始文档。document 文档同时也可以存储索引中的文档。在这种情况下, 文档参数可以被替换为以下参数: index、type、id、routing、preference、version。

在前面例子的基础上, 插入我们要 percolate 的文件索引:

请求: PUT http://127.0.0.1:9200/secisland/message/1

```
{
  "message" : "A new bonsai tree in the office"
}
```

返回值:

```
{
  "_index": "secisland",
  "_type": "message",
  "_id": "1",
  "_version": 1,
  "_shards": {"total": 2, "successful": 1, "failed": 0},
  "created": true,
  "result": "created"
}
```

Percolating 已经存在的文档，可以使用搜索索引、类型、id 等方式进行替换 document 参数进行查询，例如：

请求：GET http://127.0.0.1:9200/secisland/_search

返回值：

```
{
  "query" : {
    "percolate" : {
      "field": "query",
      "document_type" : "doctype",
      "index" : "secisland",
      "type" : "message",
      "id" : "1",
      "version" : 1
    }
  }
}
```

这个时候 document 参数被替换成了 index、type、id、version。

version 是可选的，但在某些情况下是有用的。

这个搜索的返回值和之前的返回值是一样的。

2. percolate 查询高亮显示

percolate 查询同时也支持高亮显示，例如保存两查询：

查询 1

请求：PUT http://127.0.0.1:9200/secisland/queries/1?refresh

参数：

```
{
  "query" : {
    "match" : {"message" : "brown fox"}
  }
}
```

查询 2

请求：PUT http://127.0.0.1:9200/secisland/queries/2?refresh

参数：

```
{
  "query" : {
    "match" : {"message" : "lazy dog"}
  }
}
```

高亮查询设置：

请求：GET http://127.0.0.1:9200/secisland/_search

参数:

{

返

{

"took": 7,

```
"timed out": false,
```

```
" shards": {
```

```
"total": 5,
```

```
"successful": 5,
```

```
"failed": 0
```

}

```
"hits": {
```

```
"total": 2,
```

"max score": 0.5446649,

```
"hits": [
```

{

```
"_index": "secisland",
```

```
"_type": "queries",
```

```
"_id": "2",
```

```
"_score": 0.5446649,
```

```
"_source": {
```

```
"query": {
```

```
"match": { "message": "lazy dog" }
```

}

}

```
"highlight": {
```

```
"message": [
```

"The quick brown fox jumps over the `lazy` `dog`"

]

}

```
"_index": "secisland",
```

```
"_type": "queries",
```

```
"_id": "1",
```

```

    "_score": 0.5446649,
    "_source": {
      "query": {
        "match": {"message": "brown fox"}
      }
    },
    "highlight": {
      "message": [
        "The quick <em>brown</em> <em>fox</em> jumps over the lazy dog"
      ]
    }
  }
]
}

```

A.2.4 索引的变化

当运行索引映射的时候禁止关闭或删除索引操作。

在 5.0 之前的版本，当索引正在进行映射操作的时候，关闭索引或者删除索引会导致映射失败。在 5.0 中，如果索引正在进行映射操作，则禁止请求关闭索引或者删除索引操作。但这种行为对部分映射行为和以前的方式没有变化。

由于基于磁盘的规则和默认的文档值发生了几处变化，导致 warmers 已经没有作用了，因此 warmer 和 warmer API 接口被删除。

当更新到 5.0 时候，系统会直接忽略掉定义在索引中的 warmers。

在节点统计和精简节点接口中的 OS 统计中添加了 CPU 的利用率。在返回节点统计的接口对象中增加了一个新的对象。这个对象包括 cpu 的利用率和 load_average 列。在之前的 os 对象中和 cpu 对象中移除了 load_average 列。现在 load_average 会分别输出 1 分钟，5 分钟和 15 分钟的 CPU 利用率，如果返回值中没有任何数据，表示这个节点不可用。

精简 (cat) 节点接口返回的 CPU 列被移除，取而代之的是 load_1m、load_5m、load_15m 表示 1 分钟，五分钟和十五分钟的利用率。如果返回值中没有任何数据，表示这个节点不可用。

org.elasticsearch.monitor.os.OsStats 这个类做了修改，移除了 getLoadAverage 方法。现在在 OsStats 中增加了内部类 Cpu，通过 Cpu 的 getLoadAverage 方法获取数据，返回的值不是一个 double 类型，而是一个对象类型，这个对象包括了一分钟，五分钟和十五分钟里的利用率，Cpu 方法增加了 getPercent 方法来返回当前的利用率。

索引统计中的建议统计被移到搜索统计中。作为搜索统计中的一部分。

为了减少多索引操作的困惑，在创建索引的时候不能以 + 或者 - 开头。之前建的索引还可以正常使用。

索引别名将不再支持索引路由 (index_routing)，同时也不支持搜索路由 (search_routing)。因为这两个接口并没有完全测试，同时我们希望直接对索引进行这些操作。

在 5.0 创建文档接口中用 `op_type=create` 参数将不再支持具体 id。

当调用 `_flush` 接口等待的时候 `wait_if_ongoing` 标志始终返回 `true`，因为在同一时刻有可能有别的操作对同一个分片进行刷新。如果返回 `false` 的时候，其他操作对同一个分片刷新则会导致正在进行的刷新操作被终止而不会返回任何错误。

A.2.5 聚合变化

在之前的版本中，Elasticsearch 提供了 Aggregation 缓存，如果你的数据没有变化，Elasticsearch 能够直接返回上次的缓存结果。但是有一个场景比较特殊，就是 `date` 分组，如：`from:now-30d to:now`，`now` 是一个变量，每时每刻都在变，所以 `query` 条件一直在变，这样缓存也就没有利用起来。在 Elasticsearch 5.0 中做了如下改进：

- 首先，`now` 关键字最终会被重写成具体的值。
- 其次，每个分片会根据自己的数据范围来重写查询为 `match_all` 或者 `match_none` 的查询，所以现在的查询能够被有效地缓存，并且只有个别数据有变化的分片才需要重新计算，大大提升了查询速度。

数字类型在新版本进行了重构，采用了不同的数据结构，这种结构有更好的范围查询。然而，这种结构不参与文档评分操作。为了评估在后台匹配文档的数量，数字字段需要返回到运行的查询中，这可能会导致效率的降低。

如果这个数字字段需要进行排序、范围查询、统计聚合的时候，建议用 `keyword` 字段来代替数字字段。

IP 范围聚合现在 Elasticsearch 支持 IPv6 协议，IP 地址在存储的时候用了二进制存储而不是用数字进行存储。因此 IP 聚合的时候将不再返回数字范围。

地理信息网络聚合的时候，将不再支持 `size: 0` 为零，新版本中要求 `size` 必须是大于零的数字。时间值不再支持小数，例如不再支持 `1.5h`，应该写成 `90m`。

A.3 接口相关的变化

A.3.1 文档接口变化

对文档的 `refresh` 操作将不再支持 `truthy` 和 `falsy` 值。

在创建文档接口的返回值中取消了 `created` 参数，现在返回的内容是：当创建文档的时候返回 `"operation": "create"`，当修改文档的时候返回 `"operation": "index"`。当用 `bulk` 操作的时候返回 `true`。

在删除接口的返回值中取消了 `found` 参数，取而代之的是 `"operation": "deleted"`。当返回值为 `"operation": "noop"` 的时候表示没有找到文档。当用 `bulk` 操作的时候返回 `true`。

在 5.0 之前的版本 `_reindex` 和 `_update_by_query only` 接口在 `bulk` 失败的时候仅仅返回

retried 的值。例如, 返回值:

```
{
  ...
  "retries": 10
  ...
}
```

现在当搜索失败的时候也会返回 retries 值。

```
{
  ...
  "retries": {
    "bulk": 10,
    "search": 1
  }
  ...
}
```

getAPI 接口中当最后一次刷新后, 如果文档进行了修改, 而这个时候下一次刷新还没有执行, 这个时候 get 接口就会产生问题。如果相同的文档频繁的更新将会产生冲突, 这种情况下, 可以使用 realtime=false 参数进行请求。

mget API 接口中字段 fields 被重新命名为 stored_fields。

update 和 bulk 接口中的 fields 被取消, 需要使用 _source 来加载字段。

A.3.2 CAT 接口的变化

使用 Accept 报头指定响应的 media 类型。

在精简接口 API 的返回 media 类型中, 以前的版本是通过 Accept 头的 Content-type 字段来确定的, 这和 HTTP 规范中的含义不一致, 因此新版本删除了这一特性。

_cat/nodes 接口中 host 字段被移除。因为这个字段和 ip 字段的内容是一样的。

在 recovery 接口中增加了 bytes_recovered 和 files_recovered 字段, 分别表示已恢复的字节数和文件的总数。

total_files 和 total_bytes 字段分别改名成为 files_total 和 bytes_total。

translog 字段改名为 translog_ops_recovered、translog_total 改名为 translog_ops、translog_percent 改名为 translog_ops_percent, 这三个字段的简称分别是 tor、to、top。

Changes to cat nodes API

在 nodes 接口中, m 代表 master, d 代表 data, i 代表 ingest 节点类型, 一个节点可以同时有多个角色, 当节点没有任何角色的时候表示这个节点是一个协调角色。当这个节点是集群的主节点的时候, master 列会显示 *。

A.3.3 REST API 的变化

在以前的 REST 查询中, 如果有无法识别的字符串参数将被忽略。从用户的角度来看,

这样并不合理，但现在的版本如果有不能识别的字符串参数将返回错误。

自定义 id 的长度如果超过 512 则会被拒绝。

`/_optimize` 结尾的请求将被移除，取而代之的是 `/_forcemerge`。

在 GET 方式的 HTTP 请求中 `/_forcemerge` 将不再支持，可以换成用 POST 方式的请求。

创建索引的方式只能用 PUT 方式，之前创建所以既可以用 PUT 也可以用 POST。现在只支持 PUT 方式。

判断索引是否存在的接口 `HEAD {index}/{type}` 被替换成了 `{index}/_mapping/{type}`，为了兼容性 5.0 版本还可以使用，将在 6.0 版本中移除。

在 `/_cluster/stats` 统计返回值中去掉了 mem 内存部分。

在接口 `/_cluster/state` 的 routing table 中移除了分片版本号。在集群状态中存储了分片的 id，用选择主分片的方式来代替版本信息。

节点角色信息将不再是节点属性的一部分。节点角色在节点统计的返回值中。

禁止不带引号的 JSON，此前，JSON 文档被允许有不带引号的字段的名称，这种写法不是严谨的 JSON 格式，如果在之前的 Elasticsearch 版本中有不带引号的字段，有些操作可能会报错，因此在 `jvm.options` 文件中增加了一个配置 `-Delasticsearch.json.allow_unquoted_field_names`。这个配置将在 6.0 版本中移除。

过滤接口中的 `char_filters` 参数被命名为 `char_filter`。`token_filters` 参数将被移除，用 `filter` 代替。

Delete-By-Query 插件中的 DELETE `/_query` 请求被移除，用 Delete By Query 接口代替。

PUT `/_scripts/{lang}/{id}/_create` 创建脚本索引被移除，用 stored scripts 来代替。

PUT `/_search/template/{id}/_create` 创建索引模板被移除。用 Pre-registered 模板来代替。

有些 REST 接口的结尾可以增加键值对的方式，现在这种方式被移除。

在 `_cluster/health` 以前需要大量的 `wait_for_relocating_shards` 参数，现在只需要设置 boolean 类型的 `wait_for_no_relocating_shards` 参数，如果设置为 true，表示请求将等待（直到配置超时）的集群返回之前没有分片的搬迁。默认为 false，这意味着操作不会等待。

A.4 配置相关的变化

A.4.1 参数变化

配置参数在生效前将会进行校验。在启动节点的时候，节点级别的设置和默认的设置都会被验证；动态设置的集群和索引设置在增加或者修改集群状态之前将被校验。

所有配置都注册在节点或者使用的传出客户端。客户自定义的插件的配置注册在插件加载的地方，注册的方法是 `SettingsModule` 类的 `registerSettings` 方法。

Index Level Settings

在之前的版本中,指定索引级别的设置在 `elasticsearch.yaml` 文件中配置,或者在启动命令行中配置。从 5.0 以后仅仅选择地设置在节点层生效,其他设置可以设置在具体的索引中。索引模板被在每个索引上的默认值替换。

1. 节点设置

名称设置被取消,被 `node.name` 替换, `-Dname=some_node_name` 方式制定名称将被取消。

- ☐ `node.add_id_to_custom_path` 配置参数修改名称为 `add_lock_id_to_custom_path`。
- ☐ `node.name` 的默认名称是节点 id 的前 7 个字母,节点 id 是随机产生的 UUID。
- ☐ `node.mode` 和 `node.local` 两个设置被移除。本地模式通过 `discovery.type: local` 和 `transport.type:local` 来进行配置。可以通过 `http.enabled: false` 来禁止 http 访问。

2. 节点属性设置

节点级别的属性可以通过过滤来分配,可以通过 `node.attr` 前缀设置其他节点识别或者强制意识。在之前的版本是通过在节点上设置特殊节点的属性。除了这三个 `node.master`, `node.data`, `node.ingest must` 属性外,可以通过新的 `node.attr.namespace` 命名空间移除。

3. 节点类型设置

`node.client` 设置被移除,如果节点设置了 `node.client` 节点将无法启动。可以通过分别设置 `node.master`, `node.data`, `node.ingest` 来设置节点的类型。

4. 网关设置

`gateway.format` 设置被取消。默认用 `format` 设置 `smile`。

5. transport Settings

`transport.netty.bind_host` 设置被取消,用 `transport.bind_host` 来替换。

6. 安全管理设置

安全管理 `security.manager.enabled` 设置被移除,为了授权 Elasticsearch 用户特殊的许可,需要编辑本地 java 安全策略。

7. 网络设置

设置 `network.host` 的值为 `_non_loopback_value` 表示任意选择第一接口不标记为环回。相反,可以指定地址范围(用 `_local_`, `_site_for` 制定所有的环回和私有网络地址);通过显式接口的名称,主机名或地址。

`netty.epollBugWorkaround` 被移除,在高 CPU 使用率上和早期的 JVM 版本有个问题,需要通过 `netty.epollBugWorkaround` 配置解决,但这个问题是在 Java7 中才有的,现在 Elasticsearch 5.0 要求最低使用 Java8,所以此设置被移除。

在此之前，全局线程池类型可以动态调整。线程池类型有效地控制线程池的支持队列，修改这些设置需要丰富的经验，修改不好有可能会产生比较大的风险，所以删除这些配置。现在可以为每个线程池单独调整相关的线程池参数，比如：`keep_alive`，`queue_size` 等等。

8. 线程池设置

建议线程池已经取消了，现在用搜索线程池来替换。

所有线程池前缀的设置从 `threadpool` 到 `thread_pool`。

线程池的最小参数设置从 `min` 到 `core`。

线程池的最大参数设置从 `size` 到 `max`。

队列的大小对于一个固定的线程池的设置必须 `queue_size`（所有其他的变种，以前支持的都不再支持）。

现在线程池的设置是在节点级别进行设置的。因此，它不可能通过群集设置来更新线程池。

9. 分词器设置

分词器的 `index.analysis.analyzer.default_index` 将不再支持，如果想改变索引的分词器，需要修改 `index.analysis.analyzer.default` 参数进行替代。

10. Ping settings

此前，有三种 ping 超时设置：`discovery.zen.initial_ping_timeout`，`discovery.zen.ping.timeout` 和 `discovery.zen.ping_timeout`。前两者已被移除，ping 超时设置现在只有 `discovery.zen.ping_timeout`。ping 超时的默认值是三秒。

`discovery.zen.master_election.filter_client` 和 `discovery.zen.master_election.filter_data` 被移除，新的参数用 `discovery.zen.master_election.ignore_non_master_pings` 来代替。此设置用来控制在主节点选举时候的 ping 响应，只有在极端情况下才会使用这个参数，平时一般不用配置。

11. Recovery 设置

此版本删除了 1 版本中的恢复参数：

- ☐ `index.shard.recovery.translog_size` 替换为 `indices.recovery.translog_size`。
- ☐ `index.shard.recovery.translog_ops` 替换为 `indices.recovery.translog_ops`。
- ☐ `index.shard.recovery.file_chunk_size` 替换为 `indices.recovery.file_chunk_size`。
- ☐ `index.shard.recovery.concurrent_streams` 替换为 `indices.recovery.concurrent_streams`。
- ☐ `index.shard.recovery.concurrent_small_file_streams` 替换为。
- ☐ `indices.recovery.concurrent_small_file_streams`。
- ☐ `indices.recovery.max_size_per_sec` 替换为 `indices.recovery.max_bytes_per_sec`。

如果使用以上任何这些设置，请花时间来检查配置的目的。以上所有的设置都被认为是

专家级的设置，如果绝对必要的话才可以使用。如果你已经设置了以上的任何一条在集群上请使用 update API 和设置相应的取代钥匙。

下列设置已被删除，而不需要更换。

`indices.recovery.concurrent_small_file_streams` 和 `indices.recovery.concurrent_file_streams` 回收率现在是单线程的。

12. 事务日志设置

`index.translog.flush_threshold_ops` setting 将不再被支持，为了控制事物日志的冲洗将使用 `index.translog.flush_threshold_size` 来代替。

`index.translog.fs.type` 中的 `translog` 类型将不再被支持，现在缓冲区是用 8K 缓冲区。

之前默认的事务日志是在每次 `index`, `create`, `update`, `delete`, `bulk` 请求后操作，现在在这些操作后是不进行操作了，事实上，它可能是一个性能瓶颈。现在，`index.translog.sync_interval` 不接受一个小于 100ms 的值，防止 `fsyncing` 同时启用了异步操作。特殊值 0 不再被支持。

`index.translog.interval` 已被删除。

13. Request Cache 设置

`index.cache.query.enable` 和 `indices.cache.query.size` 配置被移除，取而代之是 `index.requests.cache.enable` 和 `indices.requests.cache.size`。

`indices.requests.cache.clean_interval` 取代了 `indices.cache.clean_interval`

14. Field Data Cache 设置

`indices fielddata.cache.clean_interval` 设置已被替换为 `indices.cache.clean_interval`。

15. Allocation 设置

`cluster.routing.allocation.concurrent_recoveries` 设置已被替换为

`cluster.routing.allocation.node_concurrent_recoveries`。

16. Similarity 设置

默认的 `similarity` 已被重命名为 `classic`。

17. 索引设置

`indices.memory.min_shard_index_buffer_size` 和 `indices.memory.max_shard_index_buffer_size` 设置已删除。Elasticsearch 现在允许所有堆使用量的总和和只要低于节点的 `indices.memory.index_buffer_size` 的配置的量就可以使用分片（默认为 10% 的 JVM 堆）。

移除参数 `es.max-open-files`。设置 `es.max-open-files` 参数为 `true`，可以打印系统进程打开文件的最大个数。这个设置被取消，可以从节点信息的接口获取相同的信息，如果设置得太低，则会在启动时登录一个警告。

移除参数 `es.netty.gathering`。时间证明设置 `es.netty.gathering` 这个参数为 `true` 和 `false` 的

时候都没有什么问题。

移除参数 `es.useLinkedTransferQueue`。`es.uselinkedtransferqueue` 可以用来控制群集服务队列的实现，在集群发现过程中处理 ping 的响应。这是一个未被事实证明的设置，现已删除。

缓存并发级别设置被删除。缓存并发级别设置有两个参数 `indices.requests.cache.concurrency_level`，`indices fielddata.cache.concurrency_level` 因为他们不再适用于用于请求缓存和字段数据缓存的实现。

通过系统属性配置 Elasticsearch 进行了调整，将不再支持通过系统属性配置。

通过命令行参数设置将取消。

用过 Java 环境变量设置将取消

通过 `ES_JAVA_OPTS` 环境变量设置将取消。

现在用 `-Ename.of.setting` 来设置配置。

通过双横线设置参数将取消，例如 `--name.of.setting value.of.setting`，现在用 `-Ename.of.setting=value.of.setting` 来替换。

通过 `.properties` 文件配置设置将取消。Elasticsearch 的运行配置和日志配置将不再支持 `.properties` 文件方式（就是键值对的方式将被取消）。

18. Discovery 设置

当节点有 `network.host`、`network.bind_host`、`network.publish_host`、`transport.host`、`transport.bind_host`、`transport.publish_host` 的时候，必须设置 `discovery.zen.minimum_master_node`，因为系统被认为是在生产环境中。

19. 时实设置

`action.get.realtime` 设置将被取消。

20. 存储锁设置

`bootstrap.mlockall` 设置被改名为 `bootstrap.memory_lock`。

21. 快照设置

restoring 快照映射参数 `include_global_state` 的默认值由 `true` 改为 `false`，taking 快照的默认值还是 `true`。

不再支持代表周的时间单位 `w`。

小数的时间值（例如，0.5s）不再支持。例如，这意味着当设定超时 0.5s 将被拒绝，应该输入“500ms”。

Elasticsearch 以前版本默认允许多个节点共享相同的数据目录（50 个以上），但这种情况有时候会混乱，当同时启动多个节点的时候，第二个节点有可能看到的是一个空的目录，然后认为集群没有任何的数据节点。这种设置在笔记本上做测试是比较好的，但这并不是生产环境中常用的方式。为了更安全操作，默认 `node.max_local_storage_nodes` 被设置为 1。

22. 脚本设置

索引脚本已被存储的脚本所取代，下面的设置已被替换：

- ❑ `script.indexed` 替换为 `script.stored`。
- ❑ `script.engine.*.indexed.aggs` 替换为 `script.engine.*.stored.aggs`。
- ❑ `script.engine.*.indexed.mapping` 替换为 `script.engine.*.stored.mapping`。
- ❑ `script.engine.*.indexed.search` 替换为 `script.engine.*.stored.search`。
- ❑ `script.engine.*.indexed.update` 替换为 `script.engine.*.stored.update`。
- ❑ `script.engine.*.indexed.plugin` 替换为 `script.engine.*.stored.plugin`。

其中 * 代表的脚本语言，例如 `groovy`、`mustache`、`painless` 等。

23. 脚本模式设置

以前的脚本模式设置，例如：`"script.inline: true"`、`"script.engine.groovy.inline.aggs: false"` 等，接受广泛的“truthy”或“falsy”值。现在比较严格，只支持 `true` 和 `false` 选项。

脚本沙箱设置将被移除，之前 `script.inline` 和 `script.stored` 可以设置为 `sandbox`。现在只能设置 `set script.line: true` 或者 `script.stored: true`。

搜索设置 `index.query.bool.max_clause_count` 将取消。用 `indices.query.bool.max_clause_count` 来设置布尔子句的最大数目。

A.4.2 打包接口的变化

`Apt/yum` 安装下载地址发生了变化，从 `https://packages.elastic.co` 到 `https://artifacts.elastic.co/`。

新版本启动时间会变长。在 5.0 版本中 JVM 启动参数增加了 `-XX:+AlwaysPreTouch` 标志，这个选项将在启动的时候占用 JVM 堆的所有内存页，这样在 GC 垃圾回收时间减少内存页提交的机会。但这将增加 Elasticsearch 的启动时间，同时增加 Elasticsearch 的初始化内存空间。

1. JVM 选项

Java 虚拟机选择参数配置被放在了一个新的配置文件 `jvm.options` 中。

这意味着以前的通过环境变量设置的参数都被取消了，包括 `ES_MIN_MEM`、`ES_MAX_MEM`、`ES_HEAP_SIZE`、`ES_HEAP_NEWSIZE`、`ES_DIRECT_SIZE`、`ES_USE_IPV4`、`ES_GC_OPTS`、`ES_GC_LOG_FILE`、and `JAVA_OPTS`。

当通过解压 Elasticsearch 包的时候，默认的位置在 `config/jvm.options` 中，通过 `Debain` 或者 `RPM` 包安装后的默认配置路径在 `/etc/elasticsearch/jvm.options` 中，也可以通过设置环境变量 `ES_JVM_OPTIONS` 来改变文件的路径。

2. 用于 Windows 服务的线程堆栈大小

在之前安装成 Windows 服务时，安装脚本会配置线程的堆栈大小（这是服务守护进程

需要),但现在由于配置被移到了 `jvm.options` 文件中,则安装脚本不再配置线程的堆栈大小。在新版本中,在安装成 Windows 服务前需要修改 `jvm.options` 文件,在 32 位系统中添加 `-Xss320k` 参数,在 64 位系统中添加 `-Xss1m` 参数。

3. `/bin/bash` 文件被依赖

之前的版本,用来启动 Elasticsearch 和运行插件的命令脚本依赖 Bourne-compatible shell。在 5.0 中,通过 RPM 和 Debian 包安装需要依赖 bash shell, bash shell 的默认路径是 `/bin/bash`。

在之前配置 Elasticsearch 环境变量有两种方法,一是通过占位符语法 `${env.ENV_VAR_NAME}`,二是没有 `env` 前缀的语法 `${ENV_VAR_NAME}`,现在只保留了第二种写法,第一种写法被移除。

同样,通过 JVM 系统参数设置的 Elasticsearch 配置也被取消。

在之前的版本如果遇到内存溢出或者其他致命错误则 Elasticsearch 不会停止工作,但这样可能会产生比较严重的问题,在 5.0 后,当遇到这些问题时候,会重新启动 Elasticsearch。

A.5 脚本的变化

A.5.1 脚本和模板

系统默认的脚本语言从 Groovy 到 Painless。Painless 脚本语言有类似 Groovy 的语法,更安全,更快。从 Groovy 转到 Painless 是非常简单的。

Groovy 和 Painless 在语法上有个显著的不同在使用参数上。所有的 Painless 参数必须有一个前缀,比如:

Groovy:

```
{
  "script_score": {
    "script": {
      "lang": "groovy",
      "inline": "Math.log(_score * 2) + my_modifier",
      "params": { "my_modifier": 8 }
    }
  }
}
```

Painless (my_modifier is prefixed with params):

```
{
  "script_score": {
    "script": {
      "lang": "painless",
      "inline": "Math.log(_score * 2) + params.my_modifier",
    }
  }
}
```

```

    "params": {"my_modifier": 8}
  }
}

```

script.default_lang 的设置被移除, 因为系统不再支持默认其他脚本语言。如果要使用其他脚本语言, 需要在请求的参数中显式地指出来。

有部分已经存储的过滤查询是没有制定具体的脚本语言, 这时候默认的语言由 script.legacy.default_lang 设置。

在 1.X 版本中定义的内联脚本 / 模板将被移除。基于脚本 / 模板的索引或者文件将被删除。

脚本和字符串参数将不再被使用, 将使用脚本对象的语法来代替, 这适用于 update api、script sort、script_score function、script query、scripted_metric aggregation、script_heuristic aggregation。

内联脚本将不再被使用:

```

{
  "script_score": {
    "lang": "groovy",
    "script": "Math.log(_score * 2) + my_modifier",
    "params": {"my_modifier": 8}
  }
}

```

被替换成了下面的写法:

```

{
  "script_score": {
    "script": {
      "lang": "groovy",
      "inline": "Math.log(_score * 2) + my_modifier",
      "params": {"my_modifier": 8}
    }
  }
}

```

script 和 script_file 参数将不再有效, 由基于脚本或者模板的文件 file 来代替。

基于文件的脚本的用法将不再有效:

```

{
  "script_score": {
    "script": "calculate-score",
    "params": {"my_modifier": 8}
  }
}

```

现在是:

需要。但是，由于配置被移到了 `java.options` 文件中，因此除非你不希望调整堆栈大小，否则在 Windows 服务需要修改 `java.options` 文件。在 32 位系统中添加 `-Xss30M`，在 64 位系统中添加 `-Xss1m` 参数。

```

{
  "script_score": {
    "script": {
      "lang": "groovy",
      "file": "calculate-score",
      "params": {"my_modifier": 8}
    }
  }
}

```

`script_id` 参数将不再被使用，由 `id` 来代替。

```

{
  "script_score": {
    "script_id": "indexedCalculateScore",
    "params": {"my_modifier": 8}
  }
}

```

现在是：

```

{
  "script_score": {
    "script": {
      "id": "indexedCalculateScore",
      "lang": "groovy",
      "params": {"my_modifier": 8}
    }
  }
}

```

模板查询中的 `query` 参数将不再被使用，由 `inline` 参数代替。

```

{
  "query": {
    "template": {
      "query": {"match_{{template}}": {}},
      "params": {"template": "all"}
    }
  }
}

```

现在是：

```

{
  "query": {
    "template": {
      "inline": {"match_{{template}}": {}},
      "params": {"template": "all"}
    }
  }
}

```

搜索模板中的顶层模板 `template` 字段被替换为一致的模板 / 脚本对象语法。

```
{
  "template" : {
    "query": { "match" : { "{{my_field}}" : "{{my_value}}" } },
    "size" : "{{my_size}}"
  },
  "params" : {
    "my_field" : "foo",
    "my_value" : "bar",
    "my_size" : 5
  }
}
```

现在是:

```
{
  "inline" : {
    "query": { "match" : { "{{my_field}}" : "{{my_value}}" } },
    "size" : "{{my_size}}"
  },
  "params" : {
    "my_field" : "foo",
    "my_value" : "bar",
    "my_size" : 5
  }
}
```

索引脚本和模板

索引脚本和模板已被存储的脚本所取代, 集群 `state` 中存储的脚本或者模板来代替 `.scripts` 索引。

存储脚本的默认大小不能超过 65535 个字节。可以通过 `script.max_size_in_bytes` 参数来修改这个值。如果脚本非常大, 可以考虑用 `native scripts` 本地脚本来代替存储脚本。

以前存储在 `.scripts` 索引中的脚本将不能再使用, 现在 Elasticsearch 将在集群 `state` 中读取脚本。升级到 5.X 后, `.scripts` 索引依然存在, 如果要继续使用此脚本, 需要将存储在 `.scripts` 中的脚本迁移到集群 `state` 中。两者的语法是一样的, 不需要修改。

A.5.2 脚本迁移

1. Python 迁移脚本

下面的 Python 脚本可以用来导入索引脚本到群集 `state` 中存储:

```
from elasticsearch import Elasticsearch, helpers

es = Elasticsearch([
    {'host': 'localhost'}
])
```

```
for doc in helpers.scan(es, index=".scripts", preserve_order=True):
    es.put_script(lang=doc['_type'], id=doc['_id'], body=doc['_source'])
```

该脚本使用了官方 Elasticsearch 客户端因此你需要确保在环境中安装了客户端。

2. Perl 的迁移脚本

下面的 Perl 脚本可以用来导入索引脚本到群集 state 中存储:

```
use Search::Elasticsearch;

my $es = Search::Elasticsearch->new( nodes => 'localhost:9200');
my $scroll = $es->scroll_helper( index => '.scripts', sort => '_doc');

while (my $doc = $scroll->next) {
    $e->put_script(
        lang => $doc->{_type},
        id   => $doc->{_id},
        body => $doc->{_source}
    );
}
```

3. 验证迁移脚本

当已经通过之前的脚本或以其他方式迁移了, 可以用以下请求来验证, 例如:

```
GET _cluster/state?filter_path=metadata.stored_scripts
```

响应应该包括 .scripts 索引中的所有脚本。当证明所有的 .scripts 索引都被移走了, 可以删除 .scripts 索引。

A.5.3 脚本配置的变化

1. 脚本引擎现在只能有一种方式注册

在 5.0.0 之前, 脚本引擎可以登记多方式。Javascript 脚本语言可以登记为 "lang": "js" 或者 "lang": "javascript"。脚本引擎现在只能注册一个单一的语言, 所有的 "lang": "js" 将被替换为 "lang": "javascript"。

2. 脚本的扩展文件名称只能有一种写法

之前脚本文件的扩展名有多种写法, 比如 "js" 和 "javascript", 现在所有的扩展文件名只能有 js 这一种写法。

.javascript 文件后缀的方法将不再支持。

Update REST 接口中删除了 string 参数。

cript、script_id 和 scripting_upsert 查询参数将被移除。

A.5.4 Java 脚本的变化

1. Java API 索引脚本

所有与索引脚本交互的方法都已被删除。和存储脚本进行交互的 java API 的方法在 `clusteradminclient` 类中。如果一个字符串需要提供 `bytearray` 类应用，则提供脚本的唯一途径是通过 `bytesreference` 实现。

2. Java 传输客户

`TemplateQueryBuilder` 类将被移除。

```
TransportClient transportClient = TransportClient.builder()
    .settings(Settings.builder().put("node.name", "node"))
    .addPlugin(MustachePlugin.class)
    .build();
transportClient.addTransportAddress(
    new InetSocketAddress(new InetSocketAddress(InetAddresses.
        forString("127.0.0.1"), 9300))
);
```

在 `QueryBuilders` 类中创建 `TemplateQueryBuilder` 实例的方法被移除。`TemplateQueryBuilder` 的构造函数现在可以使用。

模板查询接口将不建议使用，在下一个版本中将移除。

在 `GeoPoint` 脚本中将移除以下方法：

```
factorDistance
factorDistanceWithDefault
factorDistance02
factorDistance13
arcDistanceInKm
arcDistanceInKmWithDefault
arcDistanceInMiles
arcDistanceInMilesWithDefault
distanceWithDefault
distanceInKm
distanceInKmWithDefault
distanceInMiles
distanceInMilesWithDefault
geohashDistanceInKm
geohashDistanceInMiles
```

用下面的方法进行代替：

```
arcDistance, arcDistanceWithDefault, planeDistance, planeDistanceWithDefault,
geohashDistance, geohashDistanceWithDefault
```

A.6 其他变化

A.6.1 分片分配的变化

在此之前，当指定数量的分配副本被发现时主分片被分配（这个数量由 `index.recovery.initial_shards` 参数指定，现在已经废弃了）。在这种情况下，如果主分片的配置只有一个副本，就是集群的定义是一个单一的分片，这意味着索引的任何分片拷贝都有可能成为主分片，甚至是一个很久以前的数据。现在通过分配分片 IDs 来解决这个问题。

分配给分片拷贝一个唯一的 IDs 作为标志。这允许集群区分有效的相同的数据和路径的多个副本，在集群启动后，只有分片的副本包含最新的数据时才可以成为主分片。

通过分配 IDs 来代替以前的版本号来识别分片的拷贝，在索引分片存储 API 中也做了相应的调整。例如：

请求：GET `http://127.0.0.1:9200/secisland/_shard_stores`

返回值：

```
{
  ...
  "0": {
    "stores": [
      {
        "sPa3OgxLSYGvQ4oPs-Tajw": {
          "name": "node_t0",
          "transport_address": "local[1]",
          "attributes": {
            "mode": "local"
          }
        },
        "allocation_id": "2iNySv_OQVePRX-yaRH_lQ",
        "legacy_version": 42,
        "allocation": "primary" | "replica" | "unused",
        "store_exception": ...
      },
      ...
    ]
  },
  ...
}
```

在返回的结果中，新的版本返回字段是 `allocation_id`，老版本返回的字段是 `version`。这个字段可用于创建 Elasticsearch 的当前版本或者在集群中有效的版本中所有分片的拷贝。`legacy_version` 字段可以理解为以前版本的 `version` 字段。

路由命令在新的版本中有两条命令，`allocate_replica` 和 `allocate_empty_primary`。我们引入了一个新的命令 `allocate_stale_primary`。`allocate_replica` 命令可以理解为以前设置 `allow_primary` 为 `false` 的情况，`allocate_empty_primary` 命令可以理解为以前设置 `allow_primary` 为

ture 的情况。

自定义路由命令 Elasticsearch 不再支持注册自定义配置命令插件。

`index.shared_filesystem.recover_on_any_node` 为 `true` 的行为进行了调整, 在之前当没有分片的副本被发现时, 有可能选择任意的节点, 但现在为了考虑平衡进行有选择地分配节点。

分片副本的发现的行为也发生了变化, 在之前如果没有满足条件的分片副本则不进行分片副本分配, 但现在可以分配分片的副本。

A.6.2 HTTP 协议的变化

HTTP 协议请求压缩始终被支持, 在之前的版本中需要设置 `http.compressed` 为 `true` 才开启压缩功能。

A.6.3 插件的变化

插件命令由 `bin/plugin` 修改为 `bin/elasticsearch-plugin`。插件的文档结构也做了修改。所有的插件文件必须在 Elasticsearch 顶级的目录中。如果你用工具生成, 这种结构会自动生成。

1. 插件隔离

隔离 (isolation) 选项已被删除。每个插件都有它自己的类加载器。

2. 网站的插件删除

网站插件已被删除。网站插件应该在 Kibana 插件中实现。

3. 多播的插件删除

已删除多播。使用单播发现, 或云发现插件。

4. 自定义查询插件的实现

自定义查询插件需要在 `QueryParser` 子类中实现 `fromXContent` 方法。插件实现自定义评分功能需要在 `scorefunctionparser` 子类中实现 `fromxcontent` 方法。

Delete-By-Query 插件已移除, 现在在 Elasticsearch 的核心中用 Delete By Query 接口来实现。当删除时间比较长的时候, 可以通过 `cancel` 接口取消。

请求: `POST http://127.0.0.1:9200/secisland/_delete_by_query`

参数:

```
{
  "query": {
    "match": {
      "message": "some message"
    }
  }
}
```

返回值:

```
{
  "took" : 147,
  "timed_out": false,
  "deleted": 119,
  "batches": 1,
  "version_conflicts": 0,
  "noops": 0,
  "retries": {"bulk": 0, "search": 0},
  "throttled_millis": 0,
  "requests_per_second": -1.0,
  "throttled_until_millis": 0,
  "total": 119,
  "failures" : []
}
```

5. 取消接口

请求: POST http://127.0.0.1:9200/_tasks/task_id:1/_cancel

attachments 插件被 ingest-attachment 插件代替

Java 系统属性设置, 在之前的 Java 系统属性设置可以在插件脚本中 -D 参数直接使用。

现在这种情况将不再允许, 现在通过 ES_JAVA_OPTS 参数进行设置。

通过 path.plugins 设置自定义插件路径将取消。

自定义脚本插件将通过 ScriptPlugin 实现, 取消了以前的通过 onModule 实现。

自定义分词插件通过 AnalysisPlugin 实现, 取消了以前的通过 onModule 实现。

自定义映射插件通过 MapperPlugin 实现, 取消了以前的通过 onModule 实现。

自定义行为插件通过 ActionPlugin 实现, 取消了以前的通过 onModule 实现。

自定义 RestHandler's 插件通过 ActionPlugin 实现, 取消了以前的通过 onModule 实现。

自定义搜索插件通过 SearchPlugin 实现, 取消了以前的通过 onModule 实现。

SearchParseElement 接口将被移除。自定义请求部分只能通过扩展 (ext) 元素实现, 通过 SearchPlugin.SearchExtSpec 提供在自定义解析器的附加部分插入自定义的内容, 在 SearchExtSpec 中实现 XContent 语法。语法解析现在在协调节点。语法解析后的结果和搜索请求的其他部分会通过传输层到数据节点, 然后存储在搜索上下文中用于以后检索。

6. 测试自定义插件

ESIntegTestCase# pluginlist 已被删除, 用 Arrays.asList 来代替, 现在它不需要所有的插件都要用 java 1.8 来实现。

在 2.X 版本索引中的元字段 _size 在聚合, 脚本和排序中不能使用, 如果需要使用这些特性, 请用 5.X 版本重建索引。

A.6.4 文件系统相关的变化

在 2.X 版本中, 索引文件的子集是通过 `mmap` 打开。在 5.X 中, 在 64 为系统中所有的索引文件都是通过 `mmap` 打开, 这可能会增加虚拟内存使用量, 但影响不大, 因为这只增加了地址空间的消耗, 其他实际内存的使用情况和 2.X 类似。

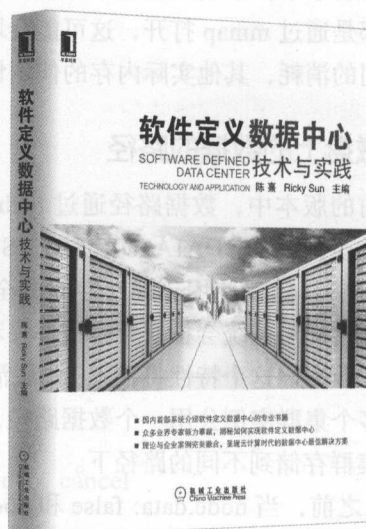
A.6.5 磁盘上的数据的路径

在之前的版本中, 数据路径通过 `path.data` 进行配置, 实际的路径包括集群名称。所以数据路径的全路径为 `$DATA_DIR/$CLUSTER_NAME/nodes/$nodeOrdinal`。在 5.0 中路径中的集群名称将不建议使用, 现在存储的全路径为 `$DATA_DIR/nodes/$nodeOrdinal`。但为了兼容性, 在启动后系统会检查集群文件夹和文件夹中的数据是否存在, 如果存在, 系统将会尽可能读取数据。这个特性将在 6.0 中取消。

如果多个集群实例公用一个数据路径, 需要在数据路径下添加集群名称, 这样就可以保证不同的集群存储到不同的路径下。

在 5.0 之前, 当 `node.data: false` 和 `node.master: false` 设置的时候, 系统是不写入任何文件到磁盘。在 5.X 中, 系统需要写入实例化节点 `ids` 标志, 需要节点来存储这些信息。因此, 所有的节点类型都将写入一个小的状态文件到其数据文件夹。

推荐阅读



深入理解大数据：大数据处理与编程实践

作者：黄宜华 等 ISBN: 978-7-111-47325-1 定价：79.00元

本书在总结多年来MapReduce并行处理技术课程教学经验和成果的基础上，与业界著名企业Intel公司的大数据技术和产品开发团队和资深工程师联合，以学术界的教学成果与业界高水平系统研发经验完美结合，在理论联系实际的基础上，在基础理论原理、实际算法设计方法以及业界深度技术三个层面上，精心组织材料编写而成。

作为国内第一本经过多年课堂教学实践总结而成的大数据并行处理和编程技术书籍，本书全面地介绍了大数据处理相关的基本概念和原理，着重讲述了Hadoop MapReduce大数据处理系统的组成结构、工作原理和编程模型，分析了基于MapReduce的各种大数据并行处理算法和程序设计的思想方法。适合高等院校作为MapReduce大数据并行处理技术课程的教材，同时也很适合作为大数据处理应用开发和编程专业技术人员的参考手册。

——中国工程院院士、中国计算机学会大数据专家委员会主任 李国杰

软件定义数据中心——技术与实践

作者：陈熹 孙宇熙 ISBN: 978-7-111-48317-5 定价：69.00元

国内首部系统介绍软件定义数据中心的专业技术书籍。

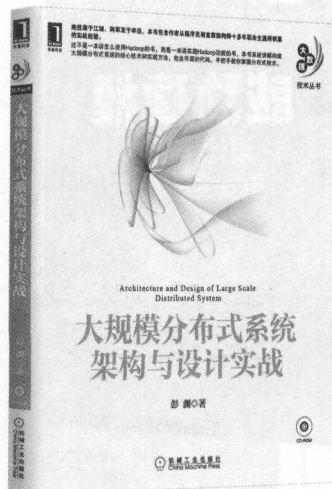
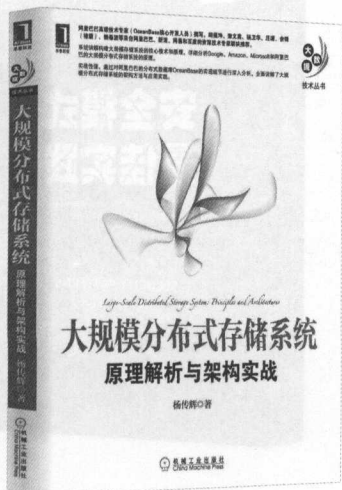
众多业界专家倾力奉献，揭秘如何实现软件定义数据中心。

理论与企业案例完美融合，呈现云计算时代的数据中心最佳解决方案。

有了以软件定义数据中心为基础的混合云，企业就可以进退有度，游刃有余，加上成功管理新的移动终端技术，可轻松进入“云移动”时代！这也是为什么软件定义数据中心最近获得大家注意的根本原因。EMC中国研究院编著的这本《软件定义数据中心：技术与实践》恰逢其时，它会给读者详细解说怎么实现软件定义数据中心。

——VMware高级副总裁，EMC中国卓越研发集团创始人 Charles Fan

推荐阅读



大规模分布式存储系统：原理解析与架构实战

作者：杨传辉 ISBN：978-7-111-43052-0 定价：59.00元

阿里巴巴高级技术专家（OceanBase核心开发人员）撰写，阳振坤、章文嵩、杨卫华、汪源、余锋（褚霸）、赖春波等来自阿里、新浪、网易和百度的资深技术专家联袂推荐

系统讲解构建大规模存储系统的核心技术和原理，详细分析Google、Amazon、Microsoft和阿里巴巴的大规模分布式存储系统的原理。

实战性强，通过对阿里巴巴的分布式数据库OceanBase的实现细节进行深入分析，完整讲解了大规模分布式存储系统的架构方法与应用实践。

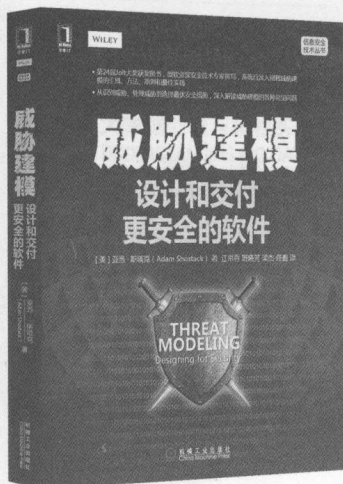
大规模分布式系统架构与设计实战

作者：彭渊 ISBN：978-7-111-45503-5 定价：59.00元

绝技源于江湖、将军发于卒伍，
本书包含作者从程序员到首席架构师十多年职业生涯所经历的实战经验

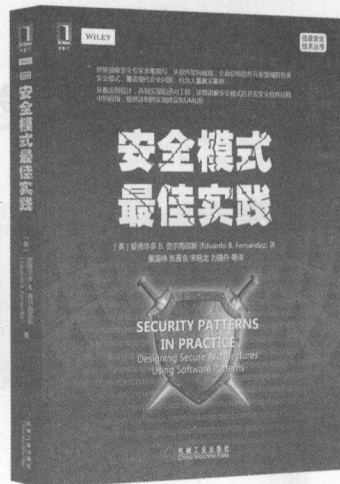
这不是一本讲怎么使用Hadoop的书，而是一本讲实现Hadoop功能的书，本书系统讲解构建大规模分布式系统的核心技术和实现方法，包含开源的代码，手把手教你掌握分布式技术。

推荐阅读



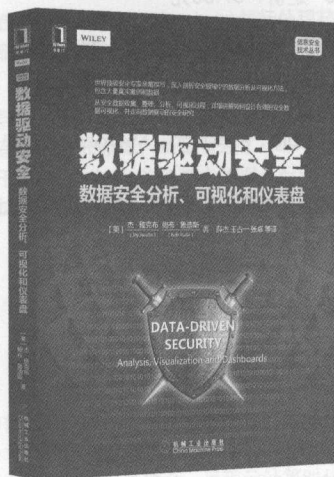
威胁建模：设计和交付更安全的软件

作者：亚当·斯塔克 ISBN：978-7-111-49807-0 定价：89.00元



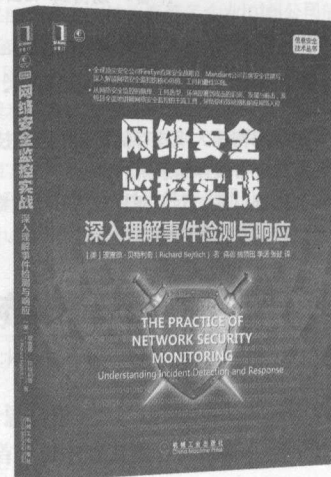
安全模式最佳实践

作者：爱德华B.费楠德 ISBN：978-7-111-50107-7 定价：99.00元



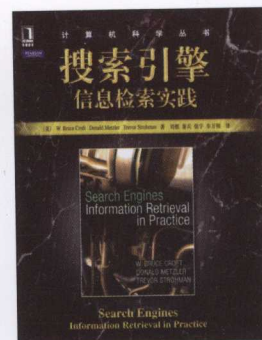
数据驱动安全：数据安全分析、可视化和仪表盘

作者：杰·雅克布等 ISBN：978-7-111-51267-7 定价：79.00元



网络安全监控实战：深入理解事件检测与响应

作者：理查德·贝特利奇 ISBN：978-7-111-49865-0 定价：79.00元



Elasticsearch技术解析与实战

Principles and Practice of Elasticsearch



本书从应用的角度深入浅出地对Elasticsearch做了全方位的剖析，从index、shard、document、cluster等基础概念到高阶的运维配置、集群优化等，从Lucene原理到Elasticsearch的高可用性实现。作者注重细节，甚至给出具体场景的参数配置，为学习Elasticsearch提供了快速进阶之路。

—— 吴树鹏 滴滴出行首席安全顾问

这可能是介绍Elasticsearch最详尽的中文参考书了，作者研究这个领域多年，有极强的实践经验。作者公司的SeciLog底层引擎也用的是Elasticsearch，该产品经过了多方实战检验。如果想深入研究Elasticsearch，这本书应是最合适的选择！

—— 张百川 游侠安全网 www.youxia.org 站长

Elasticsearch是众多开源搜索系统里的一把神器，我们的ZoomEye就使用了Elasticsearch作为搜索模块，Elasticsearch的高效、快捷、稳定给用户带来了很好的搜索体验。这本书是业内首次对Elasticsearch深入浅出进行全面介绍的原创成果，相信一定会对读者带来切实的帮助。

—— 赵伟 知道创宇CEO



投稿热线: (010) 88379604
客服热线: (010) 88379426 88361066
购书热线: (010) 68326294 88379649 68995259

华章网站: www.hzbook.com
网上购书: www.china-pub.com
数字阅读: www.hzmedia.com.cn

上架指导: 计算机/数据挖掘

ISBN 978-7-111-55327-4



9 787111 553274 >

定价: 79.00元